VECPAR

vector and
parallel
processing

20010302 181

Proceedings
Part II (June 22)

Faculdade de Engenharia
da Universidade do Porto

2000 June, 21 22 23

AQ F01-06-0980

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE 12 January 2001	3. REPORT TYPE AND DATES COVERED Conference Proceedings

4. TITLE AND SUBTITLE

VECPAR - 4th International Meeting on Vector and Parallel Processing
Part 11

5. FUNDING NUMBERS

F61775-00-WF071

6. AUTHOR(S)

Conference Committee

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

FEUP-FACULDADE DE ENGENHARIA DA Universidade do Porto
RUA DOS BRAGAS
Porto 4050-123
Portugal

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

EOARD
PSC 802 BOX 14
FPO 09499-0200

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

CSP 00-5071

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

A

13. ABSTRACT (Maximum 200 words)

The Final Proceedings for VECPAR - 4th International Meeting on Vector and Parallel Processing, 21 June 2000 - 23 June 2000, an interdisciplinary conference covering topics in all areas of vector, parallel and distributed computing applied to a broad range of research disciplines with a focus on engineering. The principal topics include: Cellular Automata, Computational Fluid Dynamics, Crash and Structural Analysis, Data Warehousing and Data Mining, Distributed Computing and Operating Systems, Fault Tolerant Systems, Imaging and Graphics, Interconnection Networks, Languages and Tools, Numerical Methods, Parallel and Distributed Algorithms, Real-time and Embedded Systems, Reconfigurable Systems, Linear Algebra Algorithms and Software for Large Scientific Problems, Computer Organization, Image Analysis and Synthesis, and Nonlinear Problems.

14. SUBJECT TERMS

EOARD, Modelling & Simulation, Parallel Computing, Distributed Computing

15. NUMBER OF PAGES

Three volumes:
1016 pages total
(plus TOC, and front matter)

16. PRICE CODE

N/A

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18
298-102

VECPAR'2000

4rd International Meeting on
Vector and Parallel Processing

2000, June 21-23

Conference Proceedings
Part II
(Thursday, June 22)

Preface

This book is part of a 3-volume set with the written versions of all invited talks, papers and posters presented at *VECPAR'2000 - 4th International Meeting on Vector and Parallel Processing*.

The Preface and the Table of Contents are identical in all 3 volumes (one for each day of the conference), numbered in sequence. Papers are grouped according to the session where they were presented.

The conference programme added up to a total of 6 plenary and 20 parallel sessions, comprising 6 invited talks, 66 papers and 11 posters.

It is our great pleasure to express our gratitude to all people that helped us during the preparation of this event. The expertise provided by the Scientific Committee was crucial in the selection of more than 100 abstracts submitted for possible presentation.

Even at the risk of forgetting some people, we would like to express our gratitude to the following people, whose collaboration went well beyond the call of duty. Fernando Jorge and Vítor Carvalho, for creation and maintenance of the conference web page; Alice Silva for the secretarial work; Dr. Jaime Villate for his assistance in organisational matters; and Nuno Sousa and Alberto Mota, for authoring the procedure for abstract submission via web.

Porto, June 2000   *The Organising and Scientific Committee Chairs*

VECPAR'2000 was held at Fundação Dr. António Cupertino de Miranda, in Porto (Portugal), from 21 to 23 June, 2000.

VECPAR is a series of conferences, on vector and parallel computing organised by the Faculty of Engineering of the University of Porto (FEUP) since 1993.

Committees

Organising Committee

A. Augusto de Sousa (Chair)
José Couto Marques (Co-chair)
José Magalhães Cruz

Local Advisory Committee

Carlos Costa
Raimundo Delgado
José Marques dos Santos
Fernando Nunes Ferreira
Lígia Ribeiro
José Silva Matos
Paulo Tavares de Castro
Raul Vidal

Scientific Committee

Sponsoring Organisations

The Organising Committee is very grateful to all sponsoring organisations for their support:

FEUP - Faculdade de Engenharia da Universidade do Porto
UP - Universidade do Porto

CMP - Câmara Municipal do Porto
EOARD - European Office of Aerospace Research and Development
FACM - Fundação Dr. António Cupertino de Miranda
FCCN - Fundação para a Computação Científica Nacional
FCG - Fundação Calouste Gulbenkian
FCT - Fundação para a Ciência e a Tecnologia
FLAD - Fundação Luso-Americana para o Desenvolvimento
ICCTI/BC - Inst. de Cooperação Científica e Tecnológica Internacional/British Council
INESC Porto - Instituto de Engenharia de Sistemas e de Computadores do Porto
OE - Ordem dos Engenheiros
Porto Convention Bureau

ALCATEL
CISCO Systems
COMPAQ
MICROSOFT
NEC European Supercomputer Systems
NORTEL Networks
SIEMENS

PART I (June 21, Wednesday)

Industrial Session 1
(June 22, Thursday, Auditorium, 11:50-12:50)

NEC European Supercomputer Systems: Vector Computing: Past Present and Future
Christian Lantwin (Manager Marketing)

CISCO Systems: 12016 Terabit System Overview
Graca Carvalho, Consulting Engineer, Advanced Internet Initiatives

Industrial Session 2
(June 22, Thursday, Room A, 11:50-12:50)

NORTEL Networks: High Speed Internet to Enable High Performance Computing
Kurt Bertone, Chief Technology Officer

COMPAQ
Title and speaker to be announced

Session 9: Numerical Methods and Parallel Algorithms
June 22, Thursday (Auditorium, 14:30-15:30)

Session 10: Linear Algebra
June 22, Thursday (Room A, 14:30-15:30)

Invited Talk
(June 22, Thursday, Auditorium, 15:30-16:30)

Session 11: Imaging
June 22, Thursday (Auditorium, 17:00-18:00)

PART III (June 23, Friday)

Solving the Quadratic 0-1 Problem

Schütz, G.[1], Pires, F. M.[2], Ruano, A. E.[2,3]

[1]**Escola Superior de Tecnologia, Universidade do Algarve**
[2]**Unidade de Ciências Exactas e Humanas, Universidade do Algarve**
[3]**Institute of Systems & Robotics**

Abstract. The quadratic 0-1 programming is a discrete optimization problem, with many important applications. Difficult graph problems can be formulated and solved as a quadratic 0-1 programming problem.
This is a NP-hard combinatorial problem very difficult to solve, even if the dimension is small. The branch-and-bound algorithms are the most used for solving exactly this sort of problems.
In this paper, based on an efficient sequential branch-and-bound algorithm for the unconstrained quadratic 0-1 programming, we study the behaviour of its parallel implementation using transputers and present some computational results. We also analyse the workload distribution among processors.

Keywords: Quadratic 0-1 programming, Branch and Bound Algorithms, Parallel Numerical Algorithms

1 Introduction

In this paper we are dealing with the unconstrained quadratic 0-1 program:

$$\min f(x) = q^T x + \tfrac{1}{2} x^T M x \qquad (1)$$
$$x \in \{0, 1\}^n$$

with $q \in \mathbb{R}^n$ and $M \in \mathbb{R}^{n \times n}$.

The quadratic 0-1 program has many interesting applications, for instance, is applied to financial analysis problems [6], CAD problems [4], circuit layout design, distributed computer networks and telecommunication networks [1]. Some difficult graph problems (like the maximum clique problem) can also be formulated and solved as a quadratic 0-1 programming problem.

As problem (1) has so many applications, it is worthwhile investing some effort in solving it. One way of solving this problem is to use a branch-and-bound algorithm. Using a branch-and-bound algorithm means to split the original problem into subproblems building a search binary tree. Each of the new subproblems must be either solved, or pruned if we can prove that it doesn't yield to a better solution. The search for good pruning techniques has been a matter of research for the last years [3], [11].

Another crucial aspect, when solving this kind of problems, is the need to produce a "good" initial solution. Some heuristics have been proposed in the last years [3], but further investigation is needed in this field, namely on how to use parallel processing to obtain a good initial guess.

Finally the search strategy is very important. The depth-first, the breadth-search and heuristic search strategies have been proposed in the context of sequential algorithms. When dealing with parallel algorithms, the search strategy must be designed accordingly to the topology and the number of processors in order to obtain a more efficient method.

The main purpose of this work is to parallelize, on transputers, a branch-and-bound algorithm for the quadratic 0-1 problem, and study its behaviour. In section 2 we summarise the branch-and-bound algorithm and the heuristic for finding the initial solution. Section 3 presents the main ideas behind the parallelization of the described algorithm. In section 4 we present some computational results using 1, 2, 4 and 8 transputers.

2 A Sequential Algorithm

The solution y* of the continuous quadratic problem constrained to the hypercube given in (1) is also a solution of the linear program [9]:

$$\min_{0 \le y \le e} \left(\nabla f(y^*)\right)^T y \qquad (2)$$

This implies that variables whose partial derivatives have fixed sign in the unitary hypercube can be fixed either to 0 or 1 according to that sign.

In order to make calculations easier, problem (1) can be formulated equivalently [9] as

$$\min f(y) = y^T A y \qquad (3)$$
$$y \in \{0, 1\}^n$$

where $a_{ii} = c_i + q_{ii} / 2$ and for $i \ne j$ $a_{ij} = q_{ij} / 2$, $i, j = 1, \dots, n$.

Without any loss of generality the matrix A can be considered to be symmetric.

It is possible to show   [8] that for problem (3) we obtain the minimum range of the partial derivatives

$$m_i \le \frac{\partial f(y)}{\partial y_i} \le M_i \text{ for } i = 1, \dots, n$$

where $m_i = 2 \sum_{\substack{j=1 \\ j \ne i}}^{n} a_{ij}^- + a_{ii}$ and $M_i = 2 \sum_{\substack{j=1 \\ j \ne i}}^{n} a_{ij}^+ + a_{ii}$

with $a_{ij}^+ = \max \{ 0, a_{ij} \}$ and $a_{ij}^- = \min \{ 0, a_{ij} \}$.

This provides an easy way of forcing variables:

a) $m_i \ge 0 \Rightarrow y_i^* = 0;$

b) $M_i \le 0 \Rightarrow y_i^* = 1.$

Hence, the gradient of the objective function characterises the difficulty of the problem, enabling to obtain smaller trees when it is possible to force more variables in the solution on the initial node. As it can be seen, from the formulas to evaluate m_i and M_i, special characteristics of the matrix A will be determinant on the number of

forced variables, namely matrices with a great number of diagonal dominant elements will provide shorter trees.

This process of forcing variables will be repeated in any node of the tree, leading to a smaller range of the partial derivatives of the non-fixed variables.

When working with branch-and-bound algorithms the main concern is to reduce the potential length of the search tree, which, at first glance, might have as much as $2^{n+1}-1$ nodes. Besides using efficient techniques to fix variables, it is also crucial to use good pruning rules. We chose to use a lower bound to the objective function as a pruning rule. If in a node the lower bound function has a value that is worse than the incumbent minimum, then that branch must be pruned, as that subproblem can never lead to a better solution. We used a lower bound function that gives a close bound to the optimum value and is very easy to evaluate and to update. To obtain this lower bound to the function f, we used the fact that its best possible value corresponds to add the rows with negative contribution to the objective function. This lower bound is easily computed from the limits of the gradient interval. Let

$F = \{i : y_i \text{ is fixed}\}$ and $\overline{F} = \{i : y_i \text{ is free}\}$, we computed the lower bound of the gradient interval for variable y_i, lb_i, as:

$$lb_i = \begin{cases} \left(\left(\sum_{\substack{j \in F \\ j \neq i}} a_{ij} x_j + \sum_{\substack{j \in F \\ j \neq i}} a_{ij}^- + \frac{1}{2} a_{ii} \right) x_i \quad, \quad i \in F \\ \sum_{\substack{j \in F \\ j \neq i}} a_{ij} x_j + \sum_{\substack{j \in F \\ j \neq i}} a_{ij}^- + \frac{1}{2} a_{ii} \qquad, \quad i \in \overline{F} \end{cases} \qquad (4)$$

and then the lower bound of the objective function given in (3) is easily computed, as it is equal to:

$$\sum_{i \in F} \left(lb_i + \frac{1}{2} a_{ii} x_i \right) + \sum_{i \in \overline{F}} \left(lb_i + \frac{1}{2} a_{ii} \right)^- $$

(5)

The update of (4) and (5) is easily and quickly done.

Another important aspect, in a branch and bound algorithm, is the order in which new subproblems are generated, that is, to choose the variable to branch in each node. Like in [9] we chose to select the variable which is most unlikely to be forced in subsequent levels of the tree. This leads to the rule of choosing the branching variable corresponding to the maximum of the values $\min\{-m_k, M_k\}$ for all the variables not yet fixed. This rule has the additional advantage of reducing the gradient range of the remaining free variables, which is favourable for fixing more variables. The value (0 or 1) assigned to that selected variable is the one that decreases lower bound function the most.

The starting point is also very important in order to obtain small trees. As a matter of fact, if the initial solution is near the optimum there is a high possibility of pruning branches earlier. There are good heuristics that allow discovering an initial solution.

Most of the times the solution obtained is close to the optimal one, if not the optimal solution itself. In this paper we used a heuristic [3] based on finding a point which is better than all its adjacent points. This point is called a local star minimum point.

3 Parallel Algorithm

Different implementations of branch-and-bound algorithms in different parallel architectures (shared memory multiprocessor, distributed memory multiprocessor and vector processors) are mentioned in the literature [2], [4], [11]. There are also references [4], [6] to the most common anomalies in parallel branch and bound algorithms, as the behaviour of such algorithms is unpredictable.

A branch and bound tree implicitly enumerates all possible solutions. Branches of the search tree are independent subproblems, so they can be evaluated in parallel. A parallel branch and bound algorithm generally splits the tree into exactly as many subproblems as there are processors. Then, each subproblem is executed in each processor, for a specified number of nodes (Maxn) of the branch-and-bound tree. When one of the processors completes its search on the tree, then an unsolved subproblem of another processor is split and assigned to that free processor. Processors also change information about new incumbents.

In this work, initially the entire problem is assigned to the root processor and the range of the gradient is used to fix all possible variables, as described above. Then the initial problem is split among all the processors. We choose the most unlikely variable x_i to be fixed in subsequent levels (as in the sequential algorithm) and split the tree into two subproblems. In the 4 transputers case, we repeated this splitting part in processor 1 and in processor 2 in order to send subproblems to processors 3 and 4, respectively. And in the 8 transputers case, processors 1, 2, 3 and 4 also repeat the splitting part in order to send subproblems to the other processors.

After a specified number of nodes, the processors communicate with its neighbours, change the incumbents solutions and send subproblems to the free processors in the same way. The algorithm stops when all processors are free, that is, when the search on the branch-and-bound tree is completed.

In this work we used two, four and eight 25 MHz INMOS transputers, on a TMB16 platform, PC hosted, with the speed of links set at 20 Mbits/sec. The programming language was AINSI C.

The topologies employed with 4 and 8 transputers are shown in figs. 1 and 2.
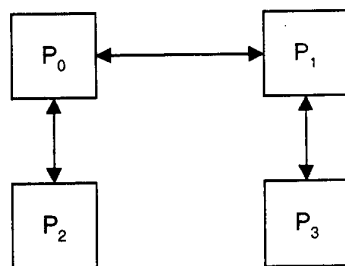
P_i = Processor i, i = 0, 1, 2, 3
Fig. 1. Topology for a 4 transputers network
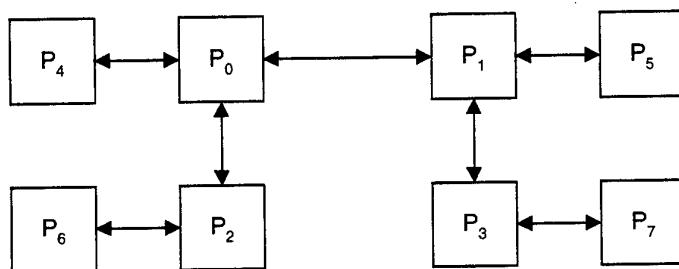
P_i = Processor i, i = 0,...,7
Fig. 2. Topology for a 8 transputers network

In the sequential branch and bound algorithm we used a depth-first strategy. In the parallel one, with these topologies, subtrees are searched in depth, one in each processor, but simultaneously, the processors, all together, perform a breadth-search in the tree because right and left branches are being searched at the same time, as shown in fig. 3 for the 8 processors case.
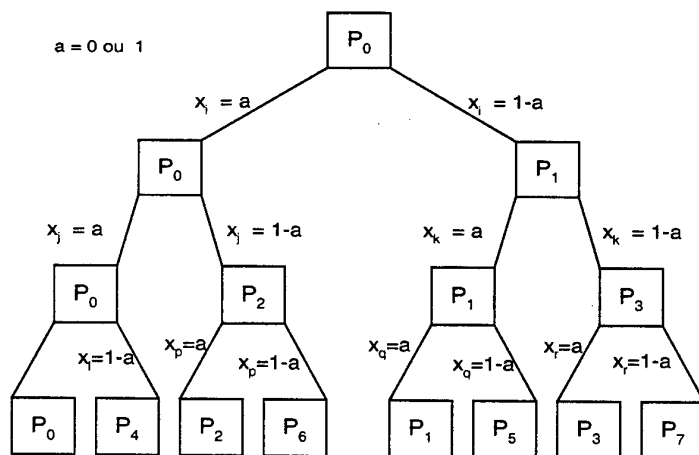
Fig. 3. Search tree for a 8 transputers network

5 Computational Results

To test the efficiency of the discrete algorithm, we have studied its behaviour when attempting to solve some problems whose matrices were taken from the Harwell-Boeing Collection (available in http://gams.nist.gov/MatrixMarket) from different sets (Structural Engineering, Partial differential equations, Power Systems Networks). These matrices are symmetric, real and not diagonal dominant. The number of variables (n) and the number of non-zero elements of the off diagonal triangular matrix (m) are described in Table 1.

Matrix name	n	m
bcsstk02	66	2145
bcsstk03	112	264
bcsstm07	420	3416
gr3030	900	3422
nos1	237	390
nos6	675	1290
nos7	729	1944
494bus	494	586

Table 1. Matrices dimensions

For each matrix problem, we randomly generated several different c^T vectors (the independent term of (3)). The optimal solution was obtained in the initial node, without any search tree, in problems: gr3030, nos6, nos7, for all the generated c^T vectors. For each one of the other problems we obtained 10 different instances. The characteristics of these test problems are described in table 2.

Probl. Set	matrix name	variables number	elements [1]		diagonal [2]				total [3] number		fixed [4] variables	
			> 0	< 0	> 0		< 0					
					min.	max.	min.	max.	min.	max.	min.	max.
1	bcsstk02	66	987	1158	46	49	17	20	2211	2211	12	16
2	bcsstk03	112	115	149	59	89	23	53	376	376	83	89
3	nos1	237	156	234	132	141	81	83	606	614	170	185
4	bcsstm07	420	2146	1270	256	257	163	164	3836	3836	382	394
5	494bus	494	0	586	286	310	184	208	1080	1080	438	463

(1) = number of non zero elements of the off diagonal triangular matrix
(2) = number of non zero elements of the diagonal plus the independent term
(3) = number of non zero elements of the triangular matrix
(4) = number of variables fixed at the initial node

Table 2. Characteristics of the test problems

As it was mentioned before, to improve the efficiency of the branch-and-bound algorithm is necessary to start with a "good" guess. The heuristic that we described before performs this task with good results. Most of the times the initial guess

obtained by the heuristic is actually the optimal solution and the branch-and-bound algorithm only is needed to confirm this optimality. The time spent with the heuristic was not included in our results because it has always been less than 0.1 seconds and is executed sequentially.

We performed some preliminary tests to determine how granularity affects speedup. In what concerns the number of nodes of the search tree, the smaller Maxn is, the better the results are. On the other hand, on what concerns speedup, the bigger the tree is, the greater Maxn should be used, and vice-versa. Nevertheless the speedup values did not vary meaningfully with Maxn. Actually, although a frequent change of information between processors reduces the search tree size, this is more time consuming and for bigger trees speedup becomes worst, as the shortness of the tree does not balance the extra increase in communication time. So, since we have an estimate, from the sequential branch and bound algorithm, of the tree dimension we decided to use accordingly values for Maxn, as shown in table 3.

Number of nodes[1]	Processors		
	2	4	8
< 1000	50	25	15
> 1000	100	50	25

(1) = Performed by the sequential branch and bound algorithm

Table 3. Values of Maxn

We began our computational study by solving the test problems sequentially. Afterwards we applied the parallel version of the algorithm with 2, 4 and 8 transputers. Table 4 summarises the obtained results.

Prob. Set		Processors															
		1		2				4				8			
		Nodes	Time	Nodes	Time	Sp	Eff	Nodes	Time	Sp	Eff	Nodes	Time	Sp	Eff
1	B(a)	310	30,43	308	17,51	1,74	0,87	306	10,02	3,04	0,76	290	11,87	5,03	0,63
	A	821	77,48	815	55,21	1,51	0,76	807	34,62	2,43	0,61	794	20,08	4,22	0,53
	W	1600	149,30	1590	114,49	1,30	0,65	446	20,72	2,04	0,51	1564	42,83	3,49	0,44
2	B(a)	259	11,46	764	27,67	1,95	0,98	764	14,74	3,67	0,92	758	7,73	7,00	0,88
	A	689	40,53	690	24,48	1,60	0,80	684	14,75	2,72	0,68	671	9,08	4,66	0,58
	W(a)	1023	60,38	250	9,91	1,16	0,58	247	6,14	1,86	0,47	239	3,78	3,03	0,38
3	B	597	174,87	3192	399,89	1,65	0,83	588	54,07	3,23	0,81	1642	56,11	7,70	0,96
	A	1467	346,82	1914	247,46	1,38	0,69	1815	125,13	2,87	0,72	1688	63,11	5,33	0,67
	W	2701	668,76	3056	358,17	1,21	0,61	3962	257,35	2,56	0,64	1691	61,74	3,79	0,47
4	B(a)	135	135,89	265	168,06	3,27	1,64	260	91,73	5,99	1,50	248	63,30	8,67	1,08
	A	385	381,46	434	248,28	1,62	0,81	380	140,29	3,58	0,90	378	101,11	5,50	0,69
	W(a)	737	715,51	265	168,06	1,06	0,53	246	95,19	1,43	0,36	319	59,85	2,27	0,28
5	B(a)	67	60,77	267	133,38	1,43	0,72	357	97,84	1,95	0,49	586	88,49	2,16	0,27
	A	254	237,79	415	207,35	1,16	0,58	478	159,79	1,47	0,37	1092	155,22	1,59	0,20
	W(a)	377	339,75	118	59,80	1,02	0,51	199	56,53	1,08	0,27	325	49,36	1,23	0,15

Sp=speedup   Eff=efficiency
B=best result   A=average   W=worst result
Time=execution time in seconds
(a)=the results for 2, 4 and 8 processors correspond to the same problem

Table 4. Summary of the results with 1, 2, 4 and 8 transputers

In this table the "best" and the "worst" results refer in the sequential case to the best and worst execution time and in the parallel case to the best and worst speedup for 2, 4 or 8 processors, so we are not always talking about the same problem. That's why we are going to focus our attention in the average line. We observe that the first three sets of problems have a similar behaviour with good values of speedup and efficiency. Set problems 4 and 5 behave in opposite directions. Set problem 4 has very good values of speedup and efficiency, reaching in some cases superlinear speedup (1, 3 and 2 cases with 2, 4 and 8 processors, respectively). Set problem 5 has poor results obtaining detrimental speedups through all cases. These results confirm the unpredictable behaviour of the branch and bound algorithms. The non-common behaviour of the problems set 4 and 5 is due to the different ways of searching the tree, in sequential and in parallel. Actually, in the sequential version, with a depth first strategy, the right branch of the tree is explored after the left one is over, while in the parallel version, with the used topology, both branches are explored simultaneously.
The good behaviour of parallel algorithm, in problem set 4, can be explained by the fact that the number of nodes of the branch-and-bound tree in the parallel version was significantly smaller than in the sequential version. As the value of the incumbent

solution is updated frequently, this allows to prune the tree earlier, mainly the left branch. So, the parallel algorithm usually needs to search fewer nodes than the sequential one, providing very good or superlinear speedups, in the cases where the sequential algorithm stops only in the middle or in the end of the right branch search.

Otherwise, in set 5, it happens that the sequential algorithm obtains the optimum in the end of the left branch search and consequently prunes the right branch of the tree, in its first iterations, while the parallel algorithm slowly decreases and updates the incumbent value, and consequently searches, in this set of problems, much more nodes, mainly in the right branch.

A reason for this behaviour may have to due with the number of positive and negative elements of the off diagonal triangular matrix of these two sets. The number of bcsstm07 matrix positive elements is almost twice the negative ones, while 494bus matrix has no positive elements (table 2).

Figs. 4 to 8 show the mean speedup obtained, for each problem set. In these graphics the dashed lines represent linear speedup.
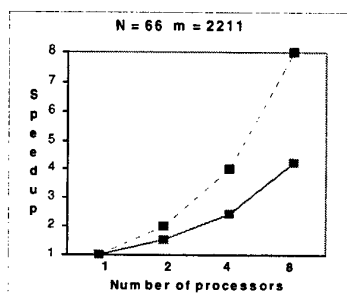
Fig. 4. Speedup for set 1
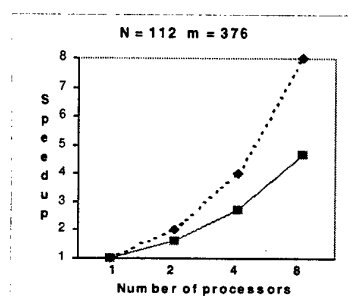
Fig. 5. Speedup for set 2
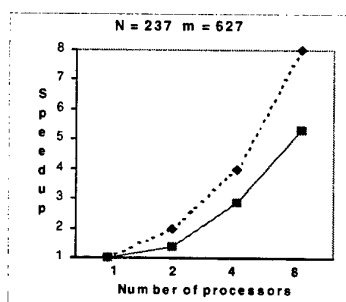
Fig. 6. Speedup for set 3
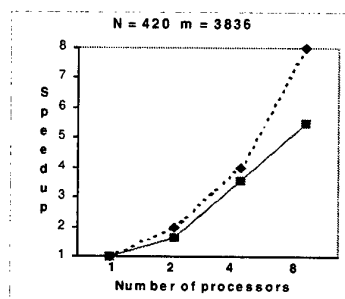
Fig. 7. Speedup for set 4
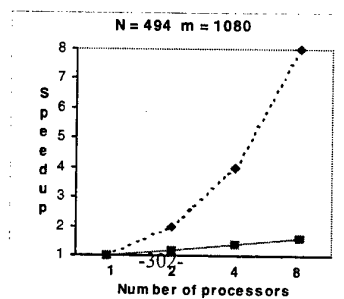
Fig. 8. Speedup for set 5

It is clear that the speedup curves are similar in figs. 4, 5 and 6. In fig. 7 speedup is almost linear. In fig. 8 speedup never gets too far from 1.

The parallel algorithm, as it can be seen in the efficiency columns and in figs. 4 to 8, performs better for 2 processors and gets worse for 8 processors. This happens because, with more processors, the communications increasing time does not balance the decrease of search nodes.

We can separate the problems with the same behaviour, with any number of processors, into two sets. One contains 16 problems which achieved good efficiencies (greater or equal than 0.6) and the other with 12 problems achieved bad efficiencies (less than 0.6). For these two groups we plotted, in fig.9, the average number of nodes searched in sequential and in parallel with 2, 4 and 8 processors. This graph confirms what was explained above about the reasons for superlinear and detrimental speedups.
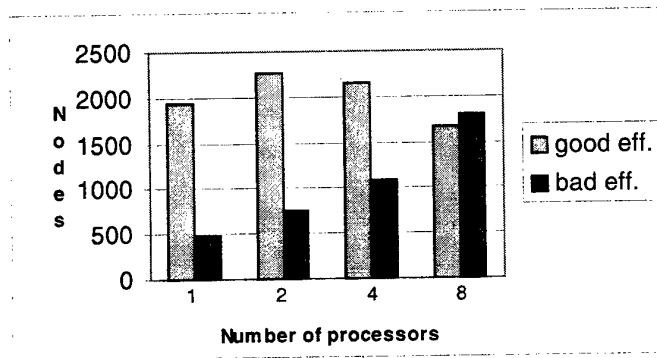
Fig. 9. Efficiency versus number of searched nodes

We also can use these two groups to study the workload distribution among processors. The average number of nodes searched by each processor is plotted in figs. 10 and 11, respectively. In fig. 12 we show the average values when considering all the problems.
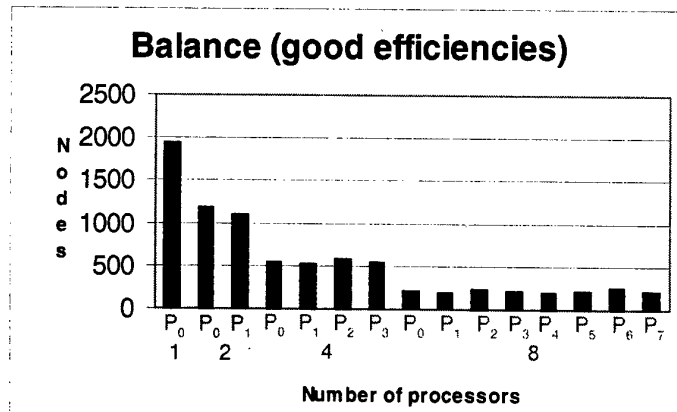
Fig. 10. Average number of nodes searched in each processor for problems with good efficiencies
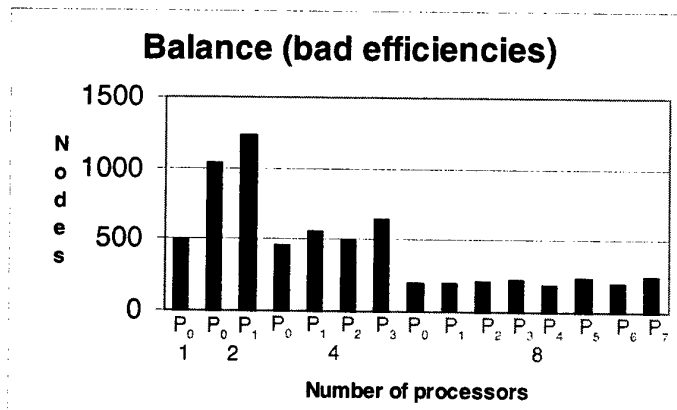
Fig. 11. Average number of nodes searched in each processor for problems with bad efficiencies
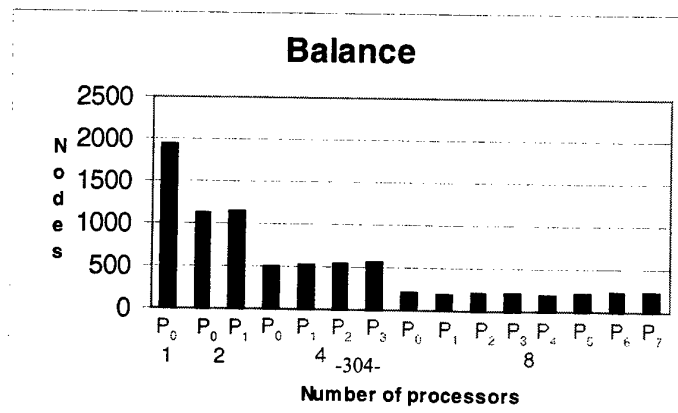
Fig. 12. Average number of nodes searched in each processor for all the
Problems

It is clear from these graphics that the workload is well balanced among processors, in almost all cases. In spite of a slight irregularity denoted in fig.11, there are no significant differences between them.

6 Conclusion

In this paper we present a study of the behaviour of a parallel algorithm for branch-and-bound search trees, implemented on a transputer network. The results obtained show that, generally, the use of parallel processing, for these kind of algorithms, is worthwhile, as significant savings in execution time can be obtained. Nevertheless, we must be aware that parallel branch and bound can also produce poor as well as superlinear speedups.

The used topology appeared quite matched for this kind of algorithms as it enables a concurrently depth and breadth tree search and achieved a well-balanced workload among processors.

We are aware of the fact that tranputers are out of date, nevertheless, we think that this study remains valid in other architectures with the same ratio between the processing capacity and communication speed.

In future, we should consider the use of different topologies, a larger number of processors, and different parallel machines, in order to solve large-scale problems in a reasonable time.

References

1. Chardaire and Sutter, *A decomposition method for quadratica zero-one programming*, Manag. Science, vol. 41, pp. 704-712 (1995).
2. Clausen, J., *Parallel branch and bound – principles and personal experiences*, in Parallel Computing in Optimization, A. Migdalas, P. Pardalos, S. Story (Eds.), Kluwer Academic Publishers (1997).
3. Faustino, A. M., *Complementaridade Linear e Aplicação em Optimização Global*, PhD Thesis, Universidade de Coimbra (1992).
4. Gendron, B., T. Crainic, *Parallel branch and bound algorithms: survey and synthesis*, Oper. Research, 42, n°6, pp1042-1066 (1994).
5. Krarup, J.; P. A. Pruzan, *Computer aided layout design*, Math. Prog. Study 9, pp.75-94 (1978).
6. Lai, T., S. Sahni, *Anomalies in parallel branch and bound algorithms*, Comm. ACM, 27, pp 119-122 (1986).
7. McBride, R. D.; J. S. Yomark, *An implicit enumeration algorithm for quadratic integer programming*, Manag. Sci. 26, pp.282-296 (1980).

8. Pardalos, P., *Construction of test problems in quadratic bivalent programming*, ACM Trans. on Math. Software, vol. 17, n°1, pp74-87 (1991).

9. Pardalos, P., *Performance of parallel branch and bound algorithms for quadratic 0-1 programming*, Tech. Rep., Comp. Science Dept., Pennsylvania State University (1990).

10. Pardalos, P., G. Rodgers, *Computational aspects of a branch and bound algorithm for quadratic 0-1 programming*, Tech. Rep., Comp. Science Dept., Pennsylvania State University (1990).

11. Pardalos, P., A. Phillips, J. Rosen, *Topics in Parallel Computing in Mathematical Programming*, Science Press (1992).

A Parallel Genetic Algorithm for Static Allocation of Real-time Tasks

Leïla Baccouche, Phd.

Laboratory of Computer Science
National Institute of Applied Science and Technology
Centre Urbain Nord B.P 676 1080 Tunis Cedex, Tunisia
Tel: 216 1 703829 Fax:216 1 704329
E-mail : Leila.baccouche@insat.rnu.tn

Abstract. This paper presents a genetic algorithm that addresses the real-time static allocation problem. In real-time systems, each task has its own timing constraints. A correct allocation is defined by schedulable tasks (no deadline is missed) on each processor. Our algorithm considers both scheduling and allocation and one major contribution of this work is that it relies on a direct problem representation and on advanced operators. Here, the problem representation clearly expresses tasks' schedules and allows to directly manipulate them. We define new genetic operators helping making choice between tasks that miss their deadlines and deciding where to move them in order to get better allocations. A parallel implementation of the algorithm is presented also a comparison with the simulated annealing algorithm. Results obtained by this algorithm are promising and presented at the end of this paper.

Keywords: real-time systems, task scheduling, static allocation, parallel genetic algorithm.

1 Introduction

Real-time scheduling is a topic where tasks have to be scheduled in order to respect timing constraints, precedence relations and resources constraints. Usually each task is described by a start time, a computing time and a deadline that must be met. Tasks can be either periodic or aperiodic, and may communicate and use resources. Most people use the term scheduling for both monoprocessor and multiprocessor. In our work, scheduling is the way to arrange a set of tasks on a single processor. We use the term of allocation to describe the way in which tasks are assigned to processors.

Task allocation, processor scheduling and communications scheduling are all NP-hard problems [4]. Thus leads to a view that they should be considered separately

and many researches handled each of these concepts independently of the others. Most related work on task allocation has mainly concentrated on maximizing the system throughput or reducing the application response time [5], [9], [8], [17], [13], [22]. These algorithms cannot be applied to real-time tasks because they rely on randomness. For each allocation obtained, these algorithms apply a post-allocation phase to determine the tasks' schedules - a task schedule is the order of tasks' execution - on the processors and to check the respect of the constraints.

Among approaches that address real-time static allocation problem, we can distinguish constructive approaches (building correct allocations) and iterative ones (modifying the current allocation in order to get a better one). Peng and Shin [18] solve the problem using two Branch&Bound algorithms (one for task allocation and another for task scheduling) for a set of communicating tasks. Hou and Shin [12] extend this work to duplicated tasks. Because of the intractability of the problem and the high cost of optimal approaches, heuristic algorithms have been developed (see the work of Ramamritham in [19], Davari and Dhall in [10] Burns and all in [21], [2] and Cheng and Agrawala in [6])

Holland developed genetic algorithms (GA) in 1975, GA are stochastic and iterative search algorithms based on the adaptive process of natural systems. They rely on the selection and the survival of the more adapted species. GA are characterized by individuals representing different allocations, a set of genetic operators used to create new individuals and a cost function to evaluate the individuals. GA have been successfully applied to a wide range of optimization problems, but only few approaches have tried to apply them to real-time task allocation problem. In [15], Kidwell presents a GA to schedule communicating tasks that can be extended to real-time scheduling. However the problem representation used does not include any consideration for scheduling. In [14], the GA is applied to tasks with precedence constraints, in [11] a GA is proposed for the job-shop scheduling problem and in [3], a GA is applied to production scheduling problems. All these works are close to our, but no algorithm from those described above can be applied or adapted to the considered model. In summary, this problem has to be adequately addressed.

In our work we develop a genetic algorithm which addresses both allocation and scheduling. We adopt a representation that reflects the nature of the problem to solve. Indeed, our representation clearly shows the tasks schedules on each processor. Besides, we develop new genetic operators specifically adapted to this new representation. When standard operators are applied randomly, tasks' scheduling is not considered in the allocation algorithm, the next step can lead to an allocation with more unschedulable tasks indeed nothing guides the way tasks are moved between processors.

The remainder of this paper is organized as follows. In section 2, we introduce the task model addressed by our algorithm. Section 3 presents the representation adopted for this problem and a method for generating initial population. Sections 4 and 5 describe the main characteristics of our genetic algorithm: the cost function that evaluates the allocations and the proposed operators. Section 6 presents

implementation results of the parallel genetic algorithm and a comparison with the simulated annealing algorithm.

2   The System and Task Model

In our model tasks have timing constraints, fault-tolerance constraints and are communicating.

Timing constraints: Each task T_i is described using 3 parameters R_i, E_i, D_i where R_i is the time at which T_i is ready and can begin its execution[1] , E_i its computing time, D_i the deadline.

Tasks allocated to a processor must be schedulable. This can be verified with the feasibility test of *EDF* the *Earliest Deadline First* approach (*EDF*)[2] [7]. In this problem the feasibility test is insufficient, the algorithm must schedule tasks in order to calculate the start time S_i at which the task is scheduled - which depends on the communications with T_j- and C_i its completion time. $C_i = S_i + E_i$.

Fault-tolerance constraints: In order to face fault-tolerance requirements, some tasks are duplicated. When a task needs some replicas, each instance must be allocated to a different processor. This way, if the processor falls down, the task execution is guaranteed on another processor.

Communications: We suppose that the communications take place after that tasks finish their computations. When calculating the start time of a task T_i, the algorithm first evaluates the delay due to communications with T_j. Indeed, the earliest date at which T_i can be scheduled depends on the reception date of all messages sent to it (which themselves depend on the completion time of sending tasks).

3   The Problem Representation in the Genetic Algorithm

A genetic algorithm consists of four steps. The generation step randomly creates a population of individuals. Each one is a potential allocation. A cost function is then applied to evaluate them. The values obtained will determine which individuals will be selected for the reproduction step. Applying genetic operators generates new individuals. The most famous are mutation and crossover. The evaluation, selection and reproduction steps are repeated until the algorithm converges.

The GA are known to have a good convergence when the following conditions are respected: (1) the coding of the individuals correctly reflects the problem to solve, (2) the individuals are in a one-to-one correspondence with search nodes (i.e. each individual corresponds to a legal search node or an allocation. Usually each

---

[1] We assume that tasks have no precedence constraints, but a programmer can express such constraints through R_i.

[2] *EDF* is a dynamic scheduling algorithm that can be applied to either periodic or aperiodic tasks. The feasibility test of *EDF* for a set of tasks is that the sum of the utilization factors of the tasks be less than or equal to one.

individual is composed of genes each one corresponding to a task and the value of the gene indicates the processor number on which it is placed. The representation used in this paper is based on the list schedules of tasks on each processor. An individual reflects for each task, on which processor it is placed, and at which position. Hence, an individual is composed of strings corresponding to the different processors and each string shows the order in which tasks will be executed on the processor. We define a correct individual as one where all tasks are represented (completeness) and only once (uniqueness). A correct individual is presented in the following figure. On processor P_1, 4 tasks are allocated and scheduled $\{T_1, T_2, T_3, T_5\}$. On processor P_2, tasks T_6 is scheduled before T_4.

$$P_1 \qquad T_1 \mid T_3 \mid T_5 \mid T_2 \mid$$
$$P_2 \qquad T_6 \mid T_4 \mid$$

Fig. 1. A problem representation with scheduling considerations

3.1 Generating the Initial Population

To generate individuals with correct schedules, we have at least to generate strings where tasks are ordered in an ascending order of deadlines. For this purpose, the set T of the N tasks to allocate is divided in classes according to deadline values.

The class 0 contains all tasks that have their deadlines in the interval [0,l] where l is the smallest deadline in T. Class 1 contains tasks whose deadlines are between]l, l+l'], class 2 contains tasks whose deadlines are between]l+l', l+2l']....l' is function of the deadline dispersion. The algorithm used to generate the initial population is described in the following:

```
Algorithm Generate-Population

GP1.  [Initialize] determine 1 and choose a value for 1'

GP2.  [Form Classes] separate the tasks according to
      their deadlines and form the classes

GP3.  [Repeat GP4 for each Pj ,j varies from 1 to M-1]

GP4.  [Allocate Tasks] Repeat for each class clk:
      randomly generate a number nb_tasks³ of tasks to pick
      from class(clk) and allocate them to Pj.

GP5.  [Allocate the remaining tasks] assign the
      remaining tasks in each class to the last processor PM
```

---

[3] Nb_tasks must be less than cardinal (clk)/M + 1 in order to allow that all the processors be served.

	R_i	Ei	Di
T1	0	1	3
T2	0	2	4
T3	0	3	4
T4	1	2	11
T5	1	1	5
T6	1	4	7
T7	4	3	8
T8	6	2	10

Individual I1 {1,2,2,1}
P1 $T_1|T_2|T_3|T_6|T_7|T_8$
P2 $T_5|T_4|$
Individual I2 {1,1,1,1}
P1 $T_1|T_2|T_6|T_8$
P2 $T_3|T_5|T_7|T_4|$
Individual I3 {0,1,2,1}
P1 $T_3|T_6|T_7|T_8|$
P2 $T_1|T_2|T_5|T_4|$

Class(0) = { T_1} Class(1) = {T_2, T_3, T_5} Class(2) = { T_6, T_7} Class(3) = { T_8, T_4}

Fig. 2. An example of population generation of 8 tasks to place on 2 processors. Here are 3 examples of individuals. The notation Individual I1 {1, 2, 2, 1} means that *nb_tasks* is respectively 1 for class 0, 2 for class 1 ....and 1 for class 3. l and l' are both equal to 3.

4   The Cost Function

The cost function of such problem is complex and has to respect the following characteristics of an allocation:
(1) Tasks meet their deadlines
(2) Replicas are allocated to different processors
These are hard constraints, besides two other criteria are to be considered:
(3) Minimize the communication costs
(4) Minimize the response time of the application

f(A) = penalty_sched(A) + penalty_replica(A) + com_cost(A) + schedule_length(A)

An allocation is correct if the characteristics (1) and (2) are respected. We define Penalty_f(A) as the sum of the first two parameters. It is equal to zero when a correct allocation is reached. Our purpose is to find a good correct solution so we try to reduce communication costs, which are the major handicap of the target machines (parallel and distributed machines with distributed memories), and the length of the allocation. The characteristics (3) and (4) help choosing between correct allocations hence they are considered only once Penalty_f(A) is equal to zero.

Calculating the penalty due to replicas: Duplicated tasks are indicated in a matrix where each row gives the index of a task and its replica. This component calculates the sum of replicas on the M processors, $nb\_T(P_j)$ is the number of tasks on a processor.

$$penalty\_replica(A) = \sum_{k=1}^{M} \sum_{i=1}^{nb\_T(P_k)} \sum_{j=i+1}^{nb\_T(P_k)} replica(i,j)$$

Replica(i,j) returns 1 if T_i is a replica of T_j and 0 otherwise.

Calculating the costs' communications: these costs depend on the amount of information to exchange between two tasks and on the distance separating the associated processors.

- $q(T_k, T_j)$ is the amount of information exchanged between T_j and T_k
- $d(P_i, P_j)$ the distance between P_i and P_j defined as the number of processors in a path from P_i to P_j minus 1.
- $com\_cost(T_i, T_j)$ is the cost of the communications between T_i and T_j. $com\_cost(T_i, T_j) = q(T_i, T_j) * d(P_i, P_j)$ when T_j is placed on P_j and T_i on P_i.
- $com\_cost(P_i, P_j)$ is the sum of all the $com\_cost(T_i, T_j)$ for all communicating tasks placed on the two processors.

$$com\_cost(A) = \sum_{j=1}^{M} \sum_{i=j+1}^{M} com\_cost(P_i, P_j)$$

Reducing these costs is equivalent to reducing the distance separating P_i and P_j. In that case, the network of processors used to calculate the distances is a logical one. Several algorithms exist for the projection of a logical network processor on a physical one. A state of the art is presented in [20].

When computing start and completion times, the algorithm takes communication delays into account in the compute of start times of receiver tasks. Let $com(T_k)$ be the set of tasks sending a message to T_k. To calculate the start time of T_k, we compute for each task T_i in $com(T_k)$ the delay necessary for the transmission of its message. T_k cannot be scheduled before its ready time is reached or before it has received all the messages.

$S_k = max \{ R_k ; max\{ C_i + com\_cost(T_k, T_j)\} \ \forall \ T_i \in com(T_k)\}$.

Calculating the penalty due to unschedulable tasks: given start and completion times, to determine $Pen(T_i)$ the penalty of a task the algorithm computes $C_i - D_i$.

We then calculate $Pen(P_j)$. We make the assumption that if a task T_i misses its deadline, and makes all tasks scheduled immediately after it, miss their deadlines too, it is sufficient to discard T_i to avoid their penalties. For this reason, we do not consider the sum of the $Pen(T_i)$ but their maximum. We are conscious that this situation occurs especially when tasks have small times between their completion times and their deadlines. Hence,

$$penalty\_sched(A) = \sum_{j=1}^{M} max(C_i - D_i), \forall T_i \in nb\_T(P_j)\}$$

Calculating the schedule length: *schedule_length(A)* is obtained after that all tasks have been allocated and scheduled on each processor :

schedule_length(A) = max { end(Pj), for j=1, ..,M} where *end(P_j) = max { C_i, ∀ T_i in nb_T(P_j) }*

If there exist an allocation with a smaller schedule length than another, surely tasks are arranged differently, and not just started earlier. Indeed the algorithm schedules tasks at the earliest date they can start.

P_1 $T_1 |_1 T_3 |_4 T_7 |_7 T_8 |_9 T_4 |_{11}$ P_1 $T_1 |_1 T_2 |_3 T_5 |_4 T_7 |_7 T_8 |_9$

P_2 $T_2 |_2 T_5 |_3 T_6 |_6$ P_2 $T_3 |_3 T_6 |_7 T_4 |_9$

Individual I_1 Individual I_2
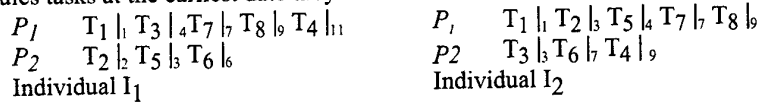
Fig. 3. A comparison of the schedule lengths of two individuals. The numbers at the right of the tasks are the tasks' completion times.

5   Design of New Genetic Operators

The main function of the genetic operators is to create new allocations, based on the current generated allocations. A new individual is created by combining or rearranging the best part of two individuals. This part can be a string or a sequence of tasks in a string.

For real-time task allocation, the genetic operators must enforce the ascending order of tasks' deadlines within a string, and respect the notions of uniqueness and completeness. The operators selected for the reproduction are the mutation and crossover. Standard mutation exchanges the value of two genes (positions) in an individual. When the representation adopted is binary, the mutation inverses the value of the gene chosen. The crossover selects split points in two individuals and exchanges the parts at the right of the split points.

Applying Crossover Operator   Applying Mutation Operator
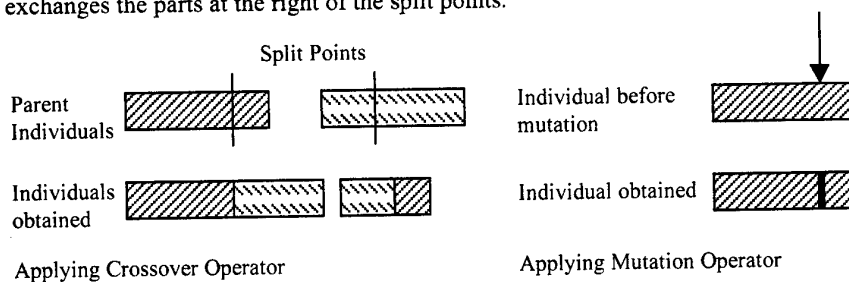
Fig. 4. Examples of standard mutation and crossover applied to a binary representation

5.1 The Crossover Operator

In our representation, we cannot randomly select split points, indeed this can lead to illegal individuals. These points must be selected according to deadlines and classes. In figure 5 crossover is applied randomly.

Individual I_1 Individual I_2

Crossover points

P1 $T_1 |_1 T_2 |_3 T_3 |_6 T_4 |_8$ P1 $T_3 |_3 T_5 |_4 T_6 |_8 T_4 |_{10}$

P2 $T_5 |_2 T_6 |_6 T_7 |_9 T_8 |_{11}$ P2 $T_1 |_1 T_2 |_3 T_7 |_6 T_8 |_8$

Individual generated I_1' Individual generated I_2'

P1 $T_1 | T_2 | T_6 | T_4 |$ P1 $T_3 | T_5 | T_3 | T_4 |$

P2 $T_5 | T_6 | T_7 | T_8 |$ P2 $T_1 | T_2 | T_7 | T_8 |$
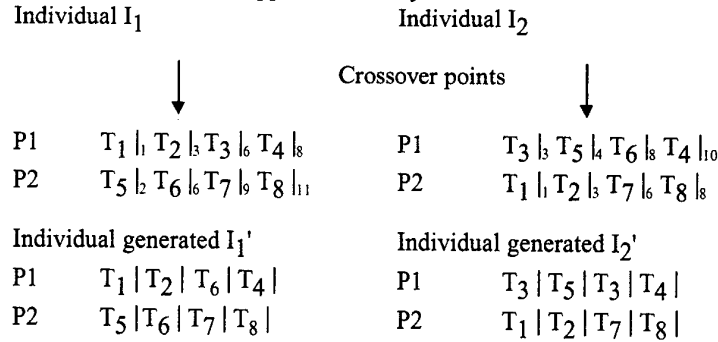
Fig. 5. Crossover creating illegal individuals. T_3 is duplicated in I_2' and absent in I_1'.

Undoubtedly, the legality of new created individuals is related to the selection of the crossover points. If tasks at the right and the left of each crossover point are of different classes, we guarantee that the generated individuals will not have tasks of the same class at both sides of the crossover point, and no duplication can be made. A similar condition was applied to tasks with precedence constraints in [14].

Theorem 1 : Let $String_k$ be the string corresponding to the kth processor on two individuals I_1 and I_2. If the crossover points between tasks i, j and i', j' satisfy the following conditions, the created individuals will be legal :

(1) $cl(T_i) < cl(T_j)$, (2) $cl(Ti') < cl(Tj')$, (3) $cl(T_i) = cl(T_i')$

$String_k$ on I_1 $T_l | T_i | T_j | T_{l'} |$ $String_k$ on I_2 $T_p | T_{i'} | T_{j'} | T_{p'} |$

The two inequality (1) and (2) enforce the notion of uniqueness. Since the task on the right of the crossover point in I2 ($T_{j'}$) is different from the one at the left on I1 (T_i) because of different classes, we assure that neither T_i nor T_j will be duplicated and tasks in $String_k$ will respect the ascending order of the classes. The equality (3) assures the completeness of the individuals. Indeed if tasks' classes could have been different, a task T_q in I_2 that satisfies class(T_i) < class(T_q) < class($T_{i'}$) will not be represented in the schedule.

To apply the crossover, we need to determine a crossover point that satisfies Theorem 1 on each string of the individuals. As the position of the crossover points can be anywhere on the string (not necessarily at the same position as in the first

individual), the individual can have strings with various numbers of tasks and with different schedule lengths. Figure 6 illustrates a correct application of the crossover.

Individual I1   Individual I2

Crossover points on the first string

P1 $T_1 |_1 T_2 |_3 T_3 |_6 T_4 |_8$ P1 $T_3 |_3 T_5 |_4 T_6 |_8 T_4 |_{10}$
P2 $T_5 |_2 T_6 |_6 T_7 |_9 T_8 |_{11}$ P2 $T_1 |_1 T_2 |_3 T_7 |_6 T_8 |_8$

Crossover points on the second string

Individual obtained I'_1 Individual obtained I'_2
P1 $T_1 |_1 T_2 |_3 T_3 |_6 T_6 |_{10} T_4 |_{12}$ P1 $T_3 |_3 T_5 |_4 T_4 |_6$
P2 $T_5 |_2 T_7 |_5 T_8 |_7$ P2 $T_1 |_1 T_2 |_3 T_6 |_7 T_7 |_{10} T_8 |_{12}$
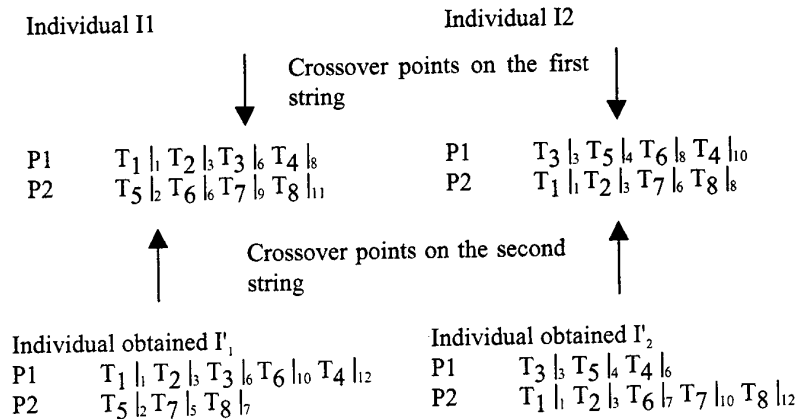
Fig. 6. A correct application of the crossover

5.2 The Mutation Operator

The standard mutation operator usually applies on two genes in an individual. Our proposed mutation operator is applied on two tasks belonging to different strings in a given individual. We not need apply mutation to the same string, indeed the scheduling algorithm determines the earliest completion time on the processor.

The proposed adaptation of the mutation operator is based on the reduction of the number of unschedulable tasks. Maybe applying the standard mutation can lead to interesting results: a randomly picked task in a schedule can reduce the schedule length on that processor and can be guaranteed in the schedule of another processor. In this way we cannot preserve the correct parts of a schedule – where all tasks meet their deadlines - but just rely on random behaviour.

Thus, the GA needs to know on each processor, the number of unschedulable tasks $nb\_Pen(P_k)$ and the first task missing its deadline $T\_Pen(P_k)$. We first search for the processor that have the highest penalty P_{hp} and the one having the smallest penalty P_{fp}. The first task missing its deadline is transferred from P_{hp} to P_{fp} or exchanged with a task having a higher deadline, depending on the value of a parameter *min_pen* (the minimum penalty to proceed to transfer). When exchange is done, we forbid the case where the task is just exchanged with a task as penalizing as the first.

A variant of this algorithm is to take the most penalizing task instead of the first task missing its deadline. More details are presented in [1].

Let us take the individuals obtained in figure 6. The created individuals I_1' and I_2' have higher penalties than their parents. On $I_{1'}$ there are 3 penalties for tasks T_3,

T_6 and T_4. $T\_pen(P_1)$ which is T_3 is transferred to P_2. On the second individual, $nb\_pen(P_2)$ is 2 with tasks T_7 and T_8. $nb\_pen(P_2)$ is not superior to min_pen so the first penalizing task T_7 is exchanged with T_4 which has a higher deadline. Both obtained allocations are correct and I_1'' has the optimal schedule_length.

I_1'' P1 $T_1 |_1 T_2 |_3 T_6 |_7 T_4 |_9$ I_2'' P1 $T_3 |_3 T_5 |_4 T_7 |_7$

     P2 $T_3 |_3 T_5 |_4 T_7 |_7 T_8 |_9$ P2 $T_1 |_1 T_2 |_3 T_6 |_7 T_8 |_9 T_4 |_{11}$

Fig. 7. Examples of mutation applied after crossover

5.3 Parallel Execution of the Genetic Algorithm

The excessive cost of GA is the main handicap for their implementation on distributed or parallel systems. In order to take advantage of the benefits of parallel systems, three parallel models for the execution of the GA were designed:

(1) First, we have the centralized model where a master processor generates the initial population and distributes the individuals on a farm of processors. Each processor (slave) executes the reproduction and selection steps and sends the best individual to the master processor. This latter proceeds to the replacement of bad individuals. One advantage is that at any moment, the best individual of a population can be known and even put in the next population. However, communication costs, which are the main handicap of these systems grow exponentially with the population size.

(2) In the second model, the population is divided in equal size subpopulations of individuals, each subpopulation being placed on a processor. Individuals are reproduced within a processor and exchanged between. This approach is interesting when the population size is greater then the number of processors. However, the parallelism internal to a subpopulation is not exploited.

(3) In the parallel model, each individual is placed on a processor. Hence, all phases from selection to replacement are done in parallel. The processors exchange their individuals with their physical neighbours. This choice reduces communication costs and fully uses the parallelism of the GA steps. For these reasons, we adopt it.

```
Algorithm Parallel-Genetic-Algorithm

PGA1. [Initialize] generate initial population and
place one individual on each processor

PGA2. [Compute penalty_f function] schedule tasks
within the local individual and calculate the
scheduling and replication penalties

PGA3. [Repeat until convergence] while a maximum number
of iterations is not reached do steps PGA3 to PGA7 on
each processor
```

```
PGA4. [Communication step] send the local individual to
neighbours and wait for theirs.

PGA5. [Selection step] select the individual among the
4 received ones that has the less value of penalty_f

PGA6. [Reproduction step]
        - apply crossover (to the local individual and the
        selected one),
        - schedule tasks within each individual and
        evaluate the penalties due to unschedulable tasks,
        - apply mutation to the individuals obtained
        - reschedule tasks and evaluate all the cost
        function in case of penalty_f equal to zero.

PGA7. [Replacement step] the local individual is
replaced if one of the generated individuals that have
a fewer cost function.
```

6 Performance Evaluation

The PGA was implemented on the supernode, a 120 transputer based machine with no shared memory. Evaluation presented was done with tasks' sets of eight and thirty to show that the algorithm has good results either with few tasks. Four benchmarks are presented:

- B1 : a graph composed of 8 tasks with no replication
- B2 : the same graph with 3 tasks replicated
- B3 : a graph of 30 tasks with no replication
- B4 : the same graph with 5 tasks replicated

For each benchmark, the PGA was run 20 times. NI is the number of iterations necessary to obtain a correct allocation and ET is the execution time given in seconds. AV is the average. Table 1 shows that the algorithm rapidly converges when no task is replicated. It is obvious that searching for the first penalizing task is less expensive than searching for the replicated tasks.

Table 1. Performance evaluation of the PGA

	NI			ET		
	Min	Max	AV	Min	Max	AV
B1	1	2	1	0.05	0.08	0.075
B2	1	20	9	0.060	0.090	0.085
B3	2	33	13	0.8	3.025	1.25
B4	4	100	40	0.7	14.75	6.035

The genetic operators are usually applied with a probability. The crossover operator we developed is applied only when split points can be found so no probability can affect the algorithm convergence with crossover. However we conceived the mutation operator to reduce the scheduling penalty and we remark when it is applied that since the first iterations the scheduling penalty is considerably reduced.
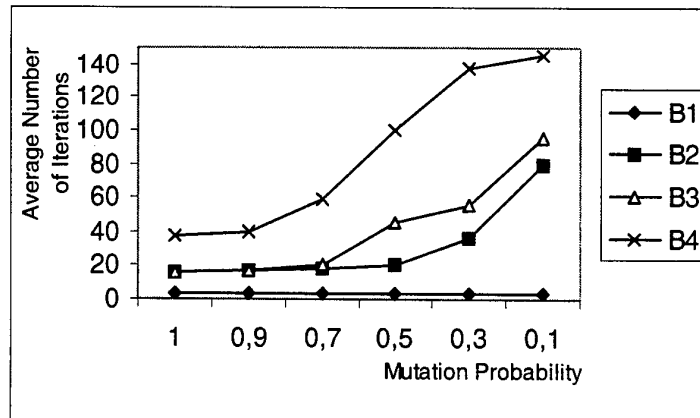
Figure 8 depicts the observed number of iterations with different mutation probabilities. A reasonable value for the mutation probability is 0.9.

Fig. 8. Observed numbers of iterations with different mutation probabilities

6.1 Comparison With the Simulated Annealing Algorithm

We applied the simulated annealing algorithm SAA to the same set of tasks, in order to show how fast our algorithm is. The SAA has been applied to solve static allocation of real-time tasks in the following works [21], [2] and [6].

The SAA uses a population of different energy states each of them corresponding to an allocation. To each state is associated a temperature which is reduced at each iteration in order to obtain the lower energy point which represents the best allocation. Neighbouring allocations are obtained by choosing a task and moving it to a randomly selected processor. The SAA cannot help in building correct allocations, as does the PGA, it can just estimate a solution using a cost function.

We had to develop a sequential version of the algorithm because the SAA cannot be paralellized easily. The SAA was applied with an initial temperature of 10 and a minimal one of 0.1.

Table 2. Comparison of the GA and the SAA for two sets of 8 and 30 tasks.

	GA		SAA	
	NI	ET	NI	ET
B1	5	0.7	72	0.8
B2	30	2.9	366	2.5
B3	420	11	945	9
B4	1200	27	5300	53

Table 2 shows that the GA does less iteration than the SAA even with a sequential version. The GA rapidly reduces the number of replicated and penalizing tasks because it can determine. However the SAA randomly moves tasks to generate new configurations. When initial temperature is high, the SAA can find the allocation that minimizes the cost function but it has no effect on when the first correct allocation is obtained. Research on guiding the SAA to move tasks is being conducted.

7   Conclusion

In this paper we considered the problem of static allocation of real-time tasks. We presented an algorithm for that purpose based on the genetic algorithms approach. We defined a problem representation that allows to directly manipulating the schedules of the tasks thus, taking into account task scheduling during allocation. We constructed efficient crossover and mutation operators that reduce the number of unschedulable tasks within an allocation, and conserve the parts of schedules that are correct in the created individuals. A parallel model for the GA was developed, and simulations conducted on a transputer-based machine are presented for various task sets. We also presented a comparison with the simulated annealing algorithm. All results show that the PGA converges rapidly.

References

1. L. Baccouche "Un mécanisme d'ordonnancement distribué de tâches temps réel" ", Phd thesis of the INPG Institute, Grenoble, France, November 95.
2. A. Burns, M. Nicholson, K. Tindell, N. Zhang " Allocating and scheduling hard real-time tasks on a point-to-point distributed system", Proc. of the Workshop on Par. and Dis. Real-time Syst. April 93.
3. R. Bruns, "Direct chromosome representation and advanced genetic operators for production scheduling", Proc. of the 5th Intern. Conf. on Gene. Algo. San Mateo 1993.

4. A. Burns, " Scheduling hard real-time systems" : a review". Softw. Engin. Journal, 6(3), 1991.
5. B. M. Carlson, L. W. Dowdy, " Static processor allocation in a soft real-time multiprocessor environment", IEEE Trans. on Par. and Distr. Syst. March 1994.
6. S-T. Cheng, A. K. Agrawala, "Allocation and scheduling of real-time periodic tasks with relative timing constraints", Tech. Report University of Maryland 1994.
7. H. Chetto, M. Chetto, " Some results of the Earliest Deadline Scheduling Algorithm ", IEEE Trans. on Soft. Eng. Vol 15, Oct 89.
8. W.W Chu, L.M Lan, " Task allocation and precedence relations for distributed real-time systems", IEEE Trans. on Comp. June 1987.
9. G-H. Chen, J-S. Yur, " A branch-and-bound-with-understimates algorithm for the task assignment problem with precedence constraint", IEEE computer 90.
10. S. Davari, S. K. Dhall, " An on-line algorithm for real-time tasks allocation", IEEE Computer 86.
11. H-L. Fang, P. Ross, D. Corne, " A promising genetic algorithm approach to a job-shop scheduling, rescheduling, and open-shop scheduling problems", Proc. of the 5th Intern. Conf. on Gene. Algo. San Mateo 1993.
12. CH. J. Hou, K.G. Shin, " Replication and allocation of task modules in distributes real-time systems", 24th Symposium on fault-tolerant computing, Austin June 94.
13. C. Houstis, " Allocation of Real-Time applications to distributed systems", Proc. Int. Conf. in Paral. Processing, August 87.
14. E. S. H. Hou, H. Ren, N. Ansari, " Efficient multiprocessor scheduling based on genetic algorithms", Dynamic, Genetic and chaotic programming 1992 John Wiley & Sons.
15. M. D. Kidwell, " Using genetic algorithms to schedule distributed tasks on a bus-based system", Proc. of the 5th Intern. Conf. on Gene. Algo. San Mateo 1993.
16. C. L. Liu, J. W. Layland, " Scheduling algorithms for multiprogramming in a hard real-time environment", Journ. of the assoc. for comp. machinery., Vol 20 Nᵢ 1, Janu. 73.
17. V.M. Lo, " Heuristics algorithms for task assignment in distributed systems" IEEE Trans. on Comp. Nov. 1988.
18. D-T. Peng, K. G. Shin, " Static allocation of periodic tasks with precedence constraints in distributed real-time systems", IEEE Proc. of the 10th Int. Conf. on Distrib. Comp. p190-198, Juin 1989.
19. K. Ramamritham, "Allocation and scheduling of complex periodic tasks", Proc. Int. Conf. on Dist. Comp. Syst. May 90.
20. E-G Talbi, " Allocation de processus sur les architectures paralèles à mémoire distribuée", PHD thesis of the INPG, grenoble, May 1993.
21. K. Tindell, A. Burns, A. Wellings " Allocating hard real-time tasks : (An NP-hard problem made easy)", Journal of Real-time systems, 1992.
22. H. Xin, Z. Hong, C. Xiyao, " Heuristic software partitioning algorithms for distributed real-time applications", IEEE Trans. on Comp. 1988.

Value prediction as a cost-effective solution to improve embedded processor performance

Silvia Del Pino, Luis Piñuel, Rafael A. Moreno and Francisco Tirado
Departamento de Arquitectura de Computadores y Automática
Universidad Complutense de Madrid. Ciudad Universitaria s/n. 28040 Madrid (Spain)
{sdelpino, lpinuel, rmoreno, ptirado}@dacya.ucm.es

Abstract. The growing market of embedded systems and applications has led to the making of more general embedded processors, with some features traditionally associated with general-purpose microprocessors. Following this trend, recent research has tried to incorporate into embedded processors the newest techniques to break down ILP limits. Value speculation is a recent technique not yet considered in the context of embedded processors, and the goal of the present work is to analyse the performance potential of this technique within this scope.

1 Introduction

Over the last few years, the increasing number of communication and multimedia applications has brought about a growing demand for high performance in embedded computing systems [1], [2], and many of the techniques for extracting Instruction-Level Parallelism (ILP), traditionally used in high performance general-purpose systems, are being applied to embedded processors [3]. The limits on the amount of extractable ILP are due to the program dependencies, and data dependencies present a particularly major hurdle. Through value speculation, it is possible to counteract data dependencies and thus increase the program's degree of parallelism.

The value prediction technique, like branch prediction, allows temporal violation of the program constraints without affecting its semantics. Based on the previous history of program execution, the hardware predicts at run-time the outcome of an instruction, which is used by the consumer instructions when the real data is not yet ready. When the true data becomes available, it is compared with the predicted value, and in the case of a mismatch, the instructions are re-executed with the correct value.

In the context of general-purpose microprocessors, the performance potential of this relatively recent technique has been shown to be significant in a number of studies [4][5]. Our intuition is that multimedia and communication programs present a more highly predictable (value) behavior than normal programs, due to the nature of both the algorithms and the input data. The objective of this work is the application of value prediction techniques in the ambit of embedded processors and the demonstration of a better efficiency within this scope.

To achieve this comparative analysis we have collected results for the integer SPEC95 and MediaBench [6] benchmarks. We used integer SPEC'95 as an evaluation benchmark in the context of general-purpose systems and MediaBench (composed of applications culled from image processing, communications and DSP applications) as

a representative benchmark set for embedded computing systems. First, we perform a predictability analysis, and we prove that the output values of the MediaBench programs are, on average, more predictable than the SPEC95 programs, using several low-cost configurations of different predictor models. However, predictability results are not enough to justify the use of extra hardware to predict values, but it is essential to prove that processor performance is also improved. So in addition, we perform detailed timing simulations in order to compare the speedup achievable by using value prediction in two typical architectures -- a high-performance embedded processor architecture and a high-performance general-purpose processor architecture --, and we prove that, using a low-cost value predictor, an embedded processor running the MediaBench programs can profit much more from value prediction than a general-purpose processor running the SPEC programs.

The paper is organised as follows. Section 2 summarises the previous work on data value prediction. Section 3 describes the experimental framework. Section 4 presents a comparative analysis of value predictability for different predictor models. Section 5 describes the two machine models used in the timing simulations and the speedup results. Finally, section 6 presents the conclusions and future work.

2   Related Work

Early work on value prediction [7] showed that instructions exhibit a new kind of locality, called *value locality*, which means that the values generated by a given static instruction tend to be repeated for a large fraction of the execution time. This property allows the data to be predictable. In a later work, Sazeides *et al.* [4] state that the predictability of a value sequence is a function of the sequence itself and the predictor used. In this way, we can find some kinds of predictable sequences, like for example the stride sequences, that do not exhibit value locality.

Most of the value predictors proposed in the literature fit into one of the following types: *Last-value predictors* (LVP), which make a prediction based on the last outcome of the same static instruction, and can correctly predict constant sequences of data. [7], [8]. *Stride predictors* (SP), which make a prediction based on the last outcome plus a constant stride, and can correctly predict arithmetic sequences of data (even constant sequences, whose stride is 0), [8], [9]. *Context-based predictors* (CBP), which learn the values that follow a particular context and make a prediction based on the last values generated by the same instruction. They can correctly predict repetitive sequences of data [4], [8]. *Hybrid predictors* (HP), which combine some of the previous predictors and include a selection mechanism, which is either hardware [8], [10], [11], or software [12]. To date, most of the implementations of these predictors have been simulated in the context of general-purpose superscalar processors using SPEC'95 as the evaluation benchmark suite. The results obtained are very promising: on average we can correctly predict about 50% of the output values of a program and obtain about a 20% improvement in speedup [10], [11], [4]. But to obtain these results, sophisticated and expensive predictors are needed, which nowadays are difficult to implement due to the current technology.

In the context of embedded processors, we can find several studies which try to improve performance by applying techniques traditionally used in the ambit of general-purpose processors. However, value prediction is a recent technique not yet

applied to these kind of processors. The reason for this lies in area restriction, a major challenge in embedded systems, which makes unfeasible the inclusion of very expensive hardware to predict values in the processor. Nevertheless, this is not the case here, since a very small predictor table is needed for this particular kind of applications as we will show later.

3 Experimental Framework

This section describes the framework employed in our research to obtain the experimental results. We performed our experiments on simulators derived from the SimpleScalar 3.0 toolset (PISA version) [13], a suite of functional and timing simulation tools.

As we mentioned above, we collected results from the integer SPEC95 and MediaBench (MB) [6] benchmarks, whose characteristics are shown in Tables 1 and 2 respectively.

Table 1. SPEC95 integer benchmark statistics.

BENCH.	DESCRIPTION	INPUT SET	# INST.	%LOAD	%INT
Compress95	Data compression	30000 e 2231	95 M	21.35	46.03
Ccl	Compiler	Ref. Input (gcc.i)	203 M	26.05	39.95
Go	Game	9 9	132 M	20.66	57.16
Ijpeg	Jpeg encoder	Train Input (specmum.ppm)	553 M	17.63	65.21
M88ksim	M88000 Simulator	Train Input	120 M	18.98	49.82
Perl	PERL interpreter	Train Input (scrabbl.in)	40 M	27.83	34.97
Li	LISP emulator	Train Input	183 M	25.90	34.74
Vortex	Data base	Train Input	2520 M	30.67	30.82

Table 2. MediaBench suite characteristics.

BENCH.	DESCRIPTION	INPUT SET	#INST.	%LOAD	%INT
Jpeg	JPEG image comp / decomp	Testimg.ppm	20 M	22.73	55.75
Mpeg	MPEG-2 video encod / decod	Rec*.YUV	1300 M	25.41	51.69
Gsm	GSM speech encod / decod	Clinton.pcm	306 M	14.88	72.47
G.721	Voice comp / decomp	Clinton.pcm	546 M	13.50	59.13
Pegwit	Public key encr / decr	Pgptest.plain	50 M	20.98	61.28
Pgp	Public key encr / decr	Pgptest.plain	153 M	17.31	67.57
Ghostscript	PostScript interpreter	Tiger.ps	1300 M	14.31	56.21
Mesa	3-D graphics library	N/A	8 M	23.22	46.10
Rasta	Speech recognition	Ex5_c1.wav	39 M	21.60	45.14
Epic	Image comp / decomp	Test_image.pgm	59 M	12.87	53.87
Adpcm	Audio encod / decod	Clinton.pcm	12 M	6.79	62.99

The *ijpeg* program belongs to both benchmark suites, but despite the name they are quite different, since not only are the library versions different but so too are the ways they used. The input files and the program parameters of the test programs are different as well. The SPEC95 version of *JPEG* was modified because the *cjped* and *djpeg* routines, for compression and decompression, required too much acceptable I/O traffic to conform to SPEC CPU guidelines; this was overcome by reading the image into a memory buffer, and processing it repeatedly with different compression settings.

The majority of the MediaBench programs are composed of two applications; compression/decompression or coding/decoding. We have combined the results for the two applications by first executing the compression or coding program and then the decompression or decoding, putting the data obtained together. The programs were compiled with the *gcc* compiler included in the tool set, using the optimization level O3. Due to time constraints, we have only simulated 100 million instructions.

4 Predictability analysis

In this section we analyze and compare value predictability for the MediaBench and SPEC95 benchmark suites. This analysis is based on the percentage of program values that can be correctly predicted. Our main purpose is to demonstrate that typical embedded applications exhibit a more predictable value behavior than normal application, especially for low-cost predictors.

As mentioned before, the predictability of a value sequence is a function of both the sequence itself and the predictor employed. Therefore, in order to accurately compare several program sets, it is necessary to carry out experiments for all the existing predictor models. Furthermore, we must consider that using idealized predictors (infinite tables) it is possible to evaluate the theoretical value predictability of programs [4], although this is not our goal. On the contrary, we want to empirically assess the program predictability by using realistic and low-cost implementations of the predictor models (limited table size). From this pragmatic analysis we should be in a position to foresee some of the performance results presented later, and we should also be able to select the most suitable value predictor for embedded processors.

4.1 Predictor models

We should first introduce the particular low-cost implementations of the predictor models which are employed in this work. In view of the fact that the last value predictions are special stride predictions (with zero stride), only stride, context-based and hybrid prediction schemes are considered. An initial analysis of each benchmark value behavior is also presented below.

Stride predictor implementation. The SP is implemented by means of a direct mapped table. The table is indexed using the least significant bits of the instruction PC. Each table entry stores the following information: the last-value produced by the instruction (32 bits), the stride between the two last outputs of the instruction (8 bits), and the confidence bits. The percentages of values correctly predicted (also called predictor efficacy) for both program suites, MediaBench (MB) and SPEC95, are shown in figure 1.

Looking at the results presented above, the first remark that should be made is that, apart from *gsm* and *pegwit*, a considerably high percentage of the MB program values could be correctly predicted by the SP (40%-50%) and very small tables are needed to achieve these results. Furthermore, except for three of the eleven programs that make up the MB suite, almost the same percentage of correct values could be obtained by using a 256-entry table or by using a 4K-entry table. On the other hand, looking at the

results for the SPEC95 benchmarks we can observe an appreciably different behavior. For most of the programs the predictor table size has a significant influence on efficacy and the results are not particularly outstanding. Nevertheless, the *m88ksim* program exhibits a particularly high value predictability, and thus appreciably raising the average results.

a) MediaBench   b) SPEC95
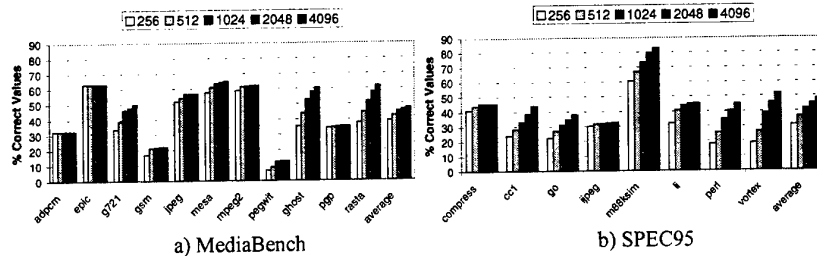
Fig 1. SP efficacy for 256, 512, 1K, 2K and 4K-entry tables.

Context-based predictor implementation. The CBP is derived from the work of Sazeides *et al.* [14] and it uses a 2-level table. The first level table, called the Value History Table (VHT) is direct mapped and it is indexed using the least significant bits of the instruction PC. This table stores an order-3 context composed by the last-value produced by the instruction (32 bits), and two strides between the 3 last outcomes produced by the instruction (8 bits each). The second level table, called the Value Prediction Table (VPT) is indexed by a hash function, which uses context information from the VHT. The VPT is responsible for storing the value prediction (32 bits) and the confidence estimation for each context. The hash function *shift-xor-fold* (also used for indexing the 2nd level table in the hybrid predictor), shown in figure 2, differs from the original one proposed by Sazeides, and significantly reduces the aliasing in the VPT (especially for small tables).
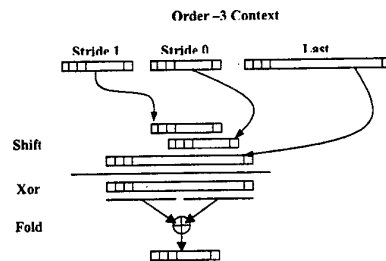
Fig 2. CBP hash function.

Figure 3 presents the efficacy results for several different CBP configurations (described in table 3). In contrast to the SP, we now remark on the significant influence of table size on predictor efficacy for both sets of benchmarks -- increasing the size of the prediction table from 256 up to 4K entries doubles the CBP efficacy for

most of the programs --. It is also important to highlight that, although for the SPEC95 suite the results of the CBP seem slightly worst than for the SP, for the MB set we appreciate a significant improvement on value predictability, especially for *gsm*, which now exhibits good predictability.

a) MediaBench   b) SPEC95
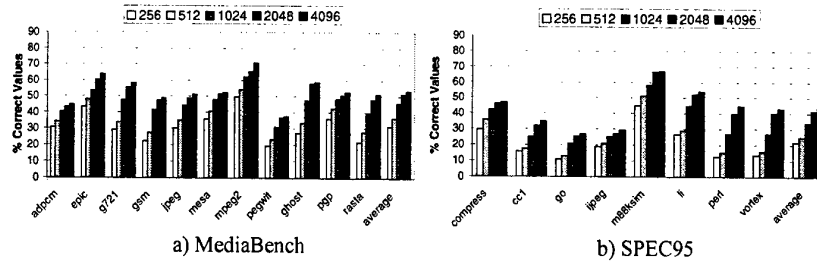
Fig 3. CBP efficacy for 256, 512, 1K, 2K and 4K-entry VPTs.

Hybrid predictor implementation. The traditional approach of implementing hybrid predictors is based on dissociated predictors and a selection mechanism. Each individual predictor produces its own prediction, and the selection logic is responsible for choosing the more suitable for the current instruction. However, when the predictable instruction sets of the predictors are highly overlapped, the hardware efficiency of this approach is low because it uses duplicated hardware for predicting the same instructions. The hybrid predictor employed in this paper is based on a previous work presented in [11]. Instead of using dissociated predictors schemes, it uses overlapped ones and a finite state machine based on value sequence classification, which decides when it is necessary to use each part of the predictor. The key idea behind this approach is to use the extra hardware only when it necessary for predicting a particular value sequence. This way for constant sequences it only uses the last-value table, for stride sequences (not constant) it uses both the last-value and stride tables, and for non-stride sequences it uses in addition the second level table. Notice that this hybrid predictor only produces a prediction at one time. The block and state diagrams of this predictor are shown in figure 4, for more details please see [11].

Several different configurations are possible for this kind of predictor, since each of the tables can be of a different size. In this work we have elected HP configurations with the same cost as the CBP. The configurations employed are described in table 3 and the HP efficacy results are shown in figure 5.

Table 3. Predictor configuration.

Predictor	Configuration					
		A	B	C	D	E
Stride	E	256	512	1024	2048	4096
Context	E_{VHT}	128	256	512	1024	1024
	E_{VPT}	256	512	1024	2048	4096
Hybrid	$E_{LAST} = E_{STRIDE}$	128	256	512	1024	1024
	E_{VPT}	256	512	1024	2048	4096

a) block structure   b) state diagram
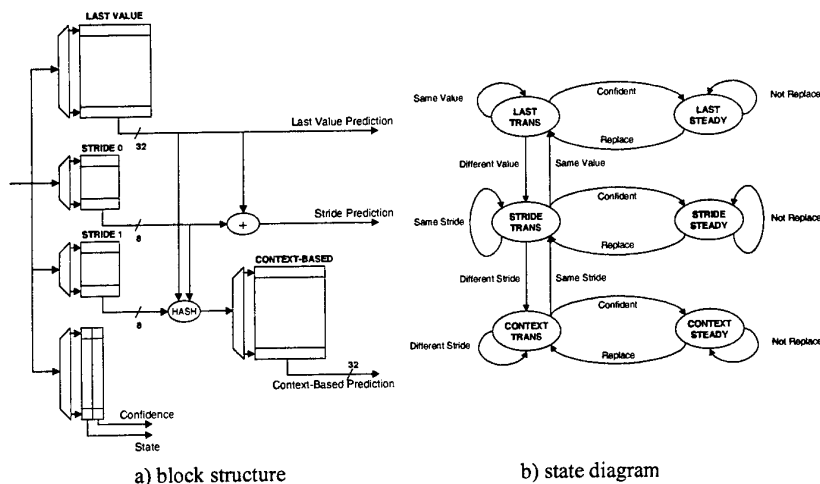
Fig 4. Hybrid predictor.

a) MediaBench   b) SPEC95
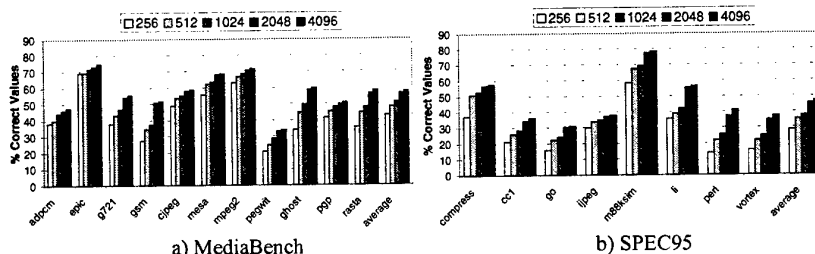
Fig 5. Hybrid predictor efficacy for 256, 512, 1K, 2K and 4K-entry VPTs.

From the results presented above we can comment that, in general, program predictability is higher for the hybrid predictor than for other predictors. Nevertheless, variations can be observed depending on the suite under consideration. For the MB suite, a remarkable increase in predictability can be noticed for all the programs, while for the SPEC95 set the previous remark is only true for a few benchmarks. In all other aspects, the behavior of the HP is similar to that of the previous predictors. With respect to the predictability of *pegwit*, although significantly better than for the SP or CBP, we observe once more that it is particularly poor compared to the other programs of the MB set (this is not true if compared to SPEC95 programs). The reason lies in the nature of the program itself. *Pegwit* is a program for public key encryption and its structure has been chosen specifically to avoid redundancy and so be resistant to cryptanalysis methods [15].

4.2 Comparative results

For the sake of highlighting the differences between both benchmark suites, we compare the average efficacy results as a function of the predictor cost. Furthermore,

these comparison also helps us to select the best predictor (i.e. best balance between efficacy and cost).

The most complex structures of the predictor are the prediction tables, and therefore we propose using the global table size as a measure of the predictor cost. Table 4 describes the formulae used to calculate the overall size of the predictors (E represents the number of table entries and N represents the number of entry field bits).

Table 4. Cost formulae.

Predictor	Global Table Size
Stride	$E * (N_{VALUE} + N_{STRIDE})$
Context	$E_{VHT} * (N_{VALUE} + 2 * N_{STRIDE}) + E_{VPT} * N_{VALUE}$
Hybrid	$E_{LAST} * N_{VALUE} + E_{VPT} * N_{VALUE} + 2 * E_{STRIDE} * N_{STRIDE}$

Figure 6 shows the average results for both sets of programs as a function of the cost. We have computed two different means in order to evaluate the uniformity of the program suite behavior: the *normal* average and the so called *realistic mean*, calculated as the arithmetic mean of all programs except those with the best and worst behaviors. In general, we observe that in the case of the MediaBench suite, both means are practically equal, but for the SPEC95 benchmarks the average is about 5% above the realistic mean. This indicates a more homogeneous behavior, in terms of value predictability, in the MediaBench set than in the SPEC95 set (which is more sensitive to the outstanding behavior of the *m88ksim* program).

a) SP   b) CBP

c) HP
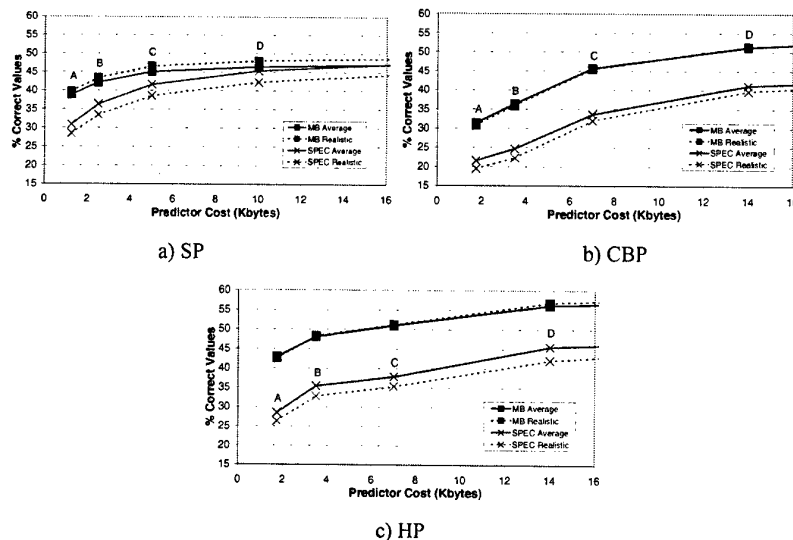
Fig 6. Comparative results for MB and SPEC benchmarks.

These results also show that predictability is higher for the MediaBench suite in all circumstances, and that the difference between both benchmark sets is more prominent for small predictor costs, decreasing as cost grows. This comparatively

high predictability of the MediaBench programs may lie in the following reasons. First, they have, on average, much more integer and less load instructions than the SPEC95 programs (see tables 2 and 3) – in fact these instructions are the most predictable instructions, as shown in [4] --. Second, MediaBench applications exhibit more loop intensive structures and more redundancy in the input data (images, voice, video...) than SPEC95 programs.

Comparing the different predictors in the case of embedded-processors, it is obvious that the hybrid predictor exhibits the best balance between efficacy and cost and hence it represents the most suitable choice. Otherwise, in the case of general-purpose processors, the HP achieves similar results to the SP.

5 Performance analysis

From the previous section we can conclude that the MediaBench suite exhibits a higher value predictability than SPEC'95. However, to justify the use of the extra value prediction hardware, it is essential to prove that the processor performance is significantly improved.

In this section we evaluate the achievable speedup from using value prediction in two typical processor architectures: a high-performance embedded processor, and a high-performance general purpose processor.

5.1 Machine model

A detailed description of all the hardware mechanisms involved in the value speculation technique is beyond the scope of the present work. We just want to briefly introduce the architecture employed in the timing simulations, which is explained in more detail in the Technical Report [16].
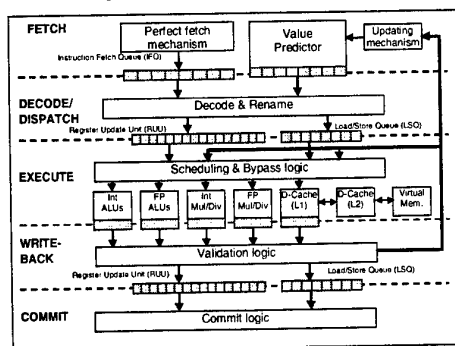
Fig. 7. Architecture Block Diagram.

Our baseline architecture, shown in figure 2, is derived from the architecture used by the SimpleScalar Out-of-Order simulator [13]. This architecture is based on the Register Update Unit (RUU) [17], which is a scheme that unifies the instruction window, the rename logic, and the reorder buffer under the same structure.

Predictor Lookup. The value predictor is accessed in parallel with the instruction fetch using the addresses of the instructions fetched in each cycle, and it provides the predicted output values (if available) of these instructions.

Scheduling policy. The *scheduling policy* firstly issues the instructions with actual operands, and thus instructions with predicted or speculative operands are issued later. Within each group, an *oldest-instruction-first* policy is used. Using this policy, speculative instructions are not issued while there are enough non-speculative instructions ready to execute, even if these non-speculative instructions are newer than the speculative ones.

Validation and misprediction recovery. The process of validation/invalidation of speculative instructions is performed during write-back. This process is performed in parallel, i.e. all the instructions within a dependence chain can be validated/invalidated in a single cycle. The instructions whose operands have been validated can commit in the next stage. On the other hand, those instructions whose operands have been invalidated must be re-executed. In view of the fact that it is not possible to check the validity and re-schedule the invalidated instructions in the same cycle, it is obvious that these instructions cannot be re-executed in the next cycle. Consequently, they are delayed one cycle in relation to normal execution.

Baseline architectures. Table 3 shows the main parameters of the two selected architectures: a 4-width embedded processor architecture and a 6-width general purpose architecture. Most of the parameters of these architectures (fetch/decode width, issue width, instruction window and L1-cache size) have been taken from two highly evolved representative commercial processors: the AMD K6-2E embedded processor core [18], and the AMD Athlon general-purpose processor core [19] (notice that fetch/decode width refers to RISC instructions). Other parameters, like functional units, have been adapted to Simplescalar Simulator, which does not support special instructions (like MMX or 3DNow). Furthermore, since value prediction significantly increases the pressure on execution units, the number of functional units and memory ports has been slightly increased in order to avoid the bottleneck in the execution stage.

Table 5. Architectural Parameters.

Configuration parameters	Embedded Processor	General-purpose Processor
Fetch/decode width	4	6
Issue width	6	9
Instruction window	24	72
Load Store Queue	12	36
# Integer ALU	4	6
# Integer Multiplier	1	2
# Floating Point ALU	4	6
# Floating Point Multiplier	1	2
# Memory Ports	2	3
L1 I Cache / L1 D Cache	32KB / 32KB	64KB / 64KB
L1 Latency	1	1
L2 Cache Size	No	4MB
L2 Latency	--	6
Memory Latency	10	10

5.2 Comparative results

In the previous section we concluded that the hybrid predictor exhibits the best cost/efficacy trade-off. This observation, along with the fact that detailed timing simulations take a long time to execute, led us to use only the hybrid predictor to show performance results.

Figure 8 shows the percentage of speedup achievable for both architectures (embedded and general purpose) and both benchmark suites (MediaBench and SPEC) using the hybrid predictor with various cost configurations (under 16 KBytes), and using a 2-bit saturating counter for confidence estimation with a confidence threshold equal to 3. Both the average and the realistic mean (eliminating the best and the worst cases) are displayed in this figure.

Two main conclusions can be drawn from this figure. First, the predictability results shown in the previous section have a direct equivalence in the performance results, since the speedup obtained for the MediaBench suite, for both architectures and all the predictor configurations, is higher than the speedup reached for the SPEC suite. Second, the difference between the average and the realistic mean curves for the SPEC benchmarks is much more prominent than the difference between the predictability curves shown in the previous section. Therefore, the sensitivity of the SPEC suite in the extreme cases has an even higher impact on speedup. This behavior is mainly due to the irregular results obtained for the m88ksim benchmark, which achieves a much higher speedup than the other benchmarks. On the other hand, MediaBench benchmarks exhibit a much more regular behavior, since the difference between the average and the realistic mean curves is of little significance.

a) Embedded Processor   b) General Purpose Processor
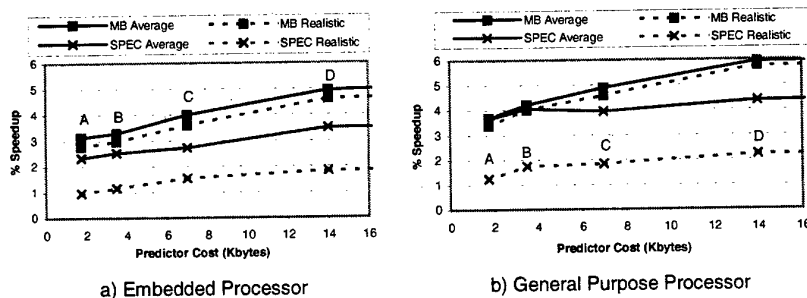
Fig. 8. % Speedup achieved with the hybrid value predictor

Figure 9 highlights the differences in the speedup achieved by using value prediction in the two habitual working situations: the embedded processor running MediaBench-like applications, and the general-purpose processor running SPEC-like applications.

Despite the general-purpose processor having wider fetch and issue, together with a larger window, the embedded processor obtains a significantly higher speedup through value prediction. These results reveal two important facts. First, as we have proved throughout this paper, typical applications of embedded systems, like MediaBench benchmarks, exhibit a higher value predictability than general-purpose applications. Second, as shown in [16], the value prediction technique gets better

speedup results when the processor uses a small to medium size instruction window. The explanation of this effect is simple. With a small to medium window size, the number of independent instructions kept in the window are not enough to cover the available issue bandwidth, hence value prediction can be efficiently exploited because it allows data dependencies to be broken, and a good number of dependent instruction to be issued in parallel. However, as the window enlarges, the number of independent instructions kept in the window also increases, and hence value prediction becomes less useful, since it is easier to find enough independent instructions in the window to feed the issue bandwidth. In view of this fact, embedded processors can benefit more from value prediction than general purpose processors, because they usually employ smaller windows due to area restrictions (24 and 72, respectively in our architectures).
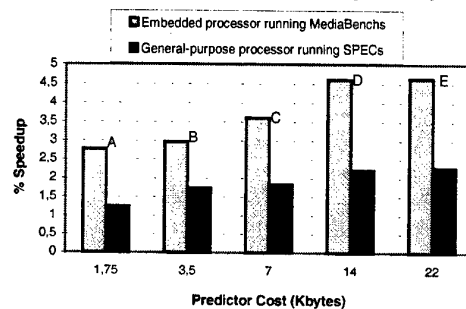
Fig. 9. Speedup achievable in habitual working situations (realistic mean)

A common question many times asked about the use of value prediction is if the extra prediction hardware spent could be better employed in other parts of the processor, which could yield a higher benefit in the overall performance -- for example increasing the L1-cache size --. With this idea in mind we performed some experiments whose results are displayed in Figure 10. This figure shows the speedup obtained by doubling the L1-cache (both the instruction and data caches) in the embedded processor and the general purpose processor (both processors running the MediaBench benchmarks), and it is compared to the speedup obtained by using a 14 Kbyte hybrid value predictor.

a) Embedded processor   b) General-purpose processor
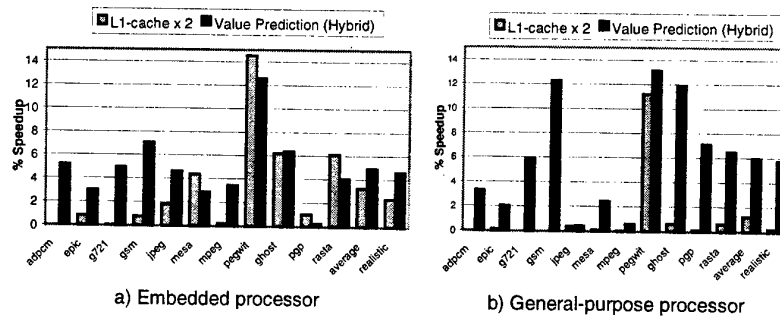
Fig. 10. Speedup obtained by doubling the L1-cache and by using value prediction

We can observe that, for both processors, the speedup achievable using the value prediction is much higher than increasing the cache size. This difference is more prominent in the general purpose processor (when executing MediaBench), since increasing the cache scarcely affects performance. Furthermore, the cost of the prediction hardware (14 Kbytes) is much lower than the cost of doubling the L1-cache (64 Kbytes in the embedded processor and, 128 Kbytes in the general purpose processor). So, we can conclude that value prediction is a profitable hardware investment for processor performance.

6   Conclusions and Future Work

The objective of this work is to apply value prediction techniques in the ambit of embedded processors and to demonstrate their higher efficiency within this scope. The main conclusions that can be drawn from this study are the following:

- Our initial intuition was verified and we have demonstrated that multimedia and communication programs present a more highly predictable value behavior than normal programs. Furthermore, a high degree of predictability can be obtained using low-cost value predictors, and therefore employing value prediction seems reasonable for this particular kind of applications.
- By means of detailed timing simulations, and using two generic high-performance architectures, one for an embedded processor and another for a general purpose processor, we have shown that the higher predictability of multimedia and communication programs has a direct impact on the performance results, since the speedup obtained for the MediaBench suite, for both architectures and all the predictor configurations is higher than the speedup attained for the SPEC suite.
- In spite of the general-purpose processor having a wider fetch and issue, as well as a larger window, the speedup achievable using value prediction in a embedded environment is significantly higher. This is due to both the higher value predictability of multimedia and communication applications and the lower instruction window used in embedded processors, which allows more efficient exploitation of value prediction.
- Finally, we have shown that the speedup obtained by using a hybrid value predictor is appreciably higher than the speedup obtained by doubling the L1-cache. These results prove that the hardware invested on value prediction is a beneficial expense for the processor performance.

Nevertheless, this work must be interpreted as a first step towards integrating value speculation into embedded processor architecture. We believe that there is considerable work to be carried out, especially in relation to performance/cost analysis, power-consumption considerations, and confidence estimation. Our future research will cover these issues, and also deepen the analysis of the hardware mechanisms involved in value speculation.

7   References

1. M.Schalett, "Trends in Embedded-Microprocessors Design", IEEE Computer, pp. 44-49, Aug. 1998.
2. J. Choquette, M. Gupta, D. McCarthy and J. Veenstra, "High-Performance RISC Microprocessors", IEEE Micro, Vol. 19, Num. 4, pp. 48-55, Jul./Aug. 99.
3. D. Connors, J. Puiatti, D. August, K. Crozier and W. Hwu, "An Architecture Framework for Introducing Predicated Execution into Embedded Microprocessors", Proc. of the 5th International Euro-par Conference, Aug. 1999.
4. Y. Sazeides, J.E. Smith, "The Predictability of Data Values," Proc. of 30th Int. Symp. on Microarchitecture (MICRO-30), pp. 248-258, Dec. 1997.
5. B. Calder, G. Reinman and D.M. Tullsen, "Selective Value Prediction", Proc. of the 26th Int. Symp. on Computer Architecture, May. 1999.
6. C. Lee and W. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", Proc. of the 30th Int. Symp. on Microarchitecture (MICRO-30), pp. 330-335, Dec. 1997.
7. M.H. Lipasti and J.P. Shen, "Exceeding the Dataflow Limit via Value Prediction," Proc. of the 29th Int. Symp. on Microarchitecture (MICRO-29), pp. 226-237, Dec. 1996.
8. K. Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors," Proc. of 30th Int. Symp. on Microarchitecture (MICRO-30), pp. 281-290, Dec. 1997.
9. T. Nakra, R. Gupta, M.L. Soffa, "Global Context-Based Value Prediction", Proc. of the 5th Int. Symp. On High Performance Computer Architecture (HPCA-5), Jan. 1999
10. B. Rychlik, J. Faisty, B. Krug, J.P. Shen, "Efficacy and Performance Impact of Value Prediction", PACT-98
11. L. Piñuel, R.A. Moreno and F.Tirado, "Implementation of hybrid context-based value predictors using value sequence classification". Proc. of the 5th International Euro-par Conference, Aug. 1999.
12. F. Gabbay and A. Mendelson, "Improving Achievable ILP Trough Value Prediction and Program Profiling", Microprocessores and Microsystems, Vol 22, n.3, Sept. 1998.
18. D. Burger and T.M. Austin. "The SimpleScalar Tool Set, Version 2.0". Technical Report CS#1342, University of Wisconsin-Madison, 1997.
19. Y. Sazeides, J.E. Smith, "Implementations of Context Based Value Predictors". Technical Report #ECE-TR-97-8, University of Wisconsin-Madison, 1997.
20. J. Daemen, L.Knudsen and V. Rijmen, "The Block Cipher SQUARE", Workshop for Fast Software Encryption, 1997
21. R. Moreno, "Using value prediction as a complexity-effective solution to improve performance", Technical Report 5/98, Dep. de Arquitectura de Computadores, Universidad Complutense de Madrid, Dec. 1998.
22. G.S. Sohi, "Instruction Issue Logic for High-Performance, Interrumpible, Multiple Functional Unit, Pipelined Computers", IEEE Trans. on Computer, 39(3), pp. 349-359, 1990
23. AMD, AMD-K6-2E Processor Data Sheet, Jan. 2000.
24. AMD, AMD Athlon Processor Data Sheet, March. 2000.

Parallel Pole Assignment of Single-Input Systems*

Maribel Castillo[1], Enrique S. Quintana-Ortí[1], Gregorio Quintana-Ortí[1], and Vicente Hernández[2]

[1] Depto. de Informática, Universidad Jaume I, 12080–Castellón, Spain; {castillo,quintana,gquintan}@inf.uji.es. Tel.: +34-964-728000. Fax: +34-964-728435.
[2] Depto. de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 46071–Valencia, Spain; vhernand@dsic.upv.es. Tel.: +34-96-3877350. Fax: +34-96-3877359.

Abstract. We present a parallelization of Petkov, Christov, and Konstantinov's algorithm for the pole assignment problem of single-input systems. Our new implementation is specially appropriate for current high performance processors and shared memory multiprocessors and obtains a high performance by reordering the access pattern, while maintaining the same numerical properties.

The experimental results on two different platforms (SGI PowerChallenge and SUN Enterprise) report a higher performance of the new implementation over traditional algorithms.

Topics: Numerical methods, parallel and distributed algorithms.

1 Introduction

Consider the continuous, time-invariant linear system defined by

$$\dot{x}(t) = Ax(t) + Bu(t), \quad x(0) = x_0,$$

with n states, in vector $x(t)$, and m inputs, in vector $y(t)$. Here, A is the $n \times n$ state matrix, and B is the $n \times m$ input matrix.

In the design of linear control systems, $u(t)$ is used to control the behaviour of the system. Specifically, the control

$$u(t) = -Fx(t),$$

where F is an $m \times n$ feedback matrix, is used to modify the properties of the closed-loop system

$$\dot{x}(t) = (A - BF)x(t).$$

The problem of finding an appropriate feedback F is referred to as the *problem of synthesis of a state regulator* [11]. In some applications, e.g., for

---

asymptotic stability [4,11], F can be chosen so that the eigenvalues of the closed-loop matrix are in the open left-half complex plane.

In this paper we are interested in the *pole assignment problem* of single-input systems ($m = 1$ and $B = b$ is a vector), or PAPSIS, which consists in the determination of a feedback vector $F = f$, such that the poles of the closed-loop system are allocated to a pre-specified set $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ [4]. This problem has a solution (unique in the single-input case) if and only if the system is controllable [15]. We assume hereafter that this condition is satisfied.

A survey of existing algorithms for the pole assignment problem can be found, e.g., in [4–6,11,14]. Among these, methods based on the Schur form of the closed-loop state matrix [6,9,10] are numerically stable [3,7].

In [2] we apply block-partitioned techniques to obtain efficient implementations of Miminis and Paige's algorithm for PAPSIS [6]. In this paper we apply similar techniques to obtain LAPACK-like [1] block-partitioned variants and parallel implementations of Petkov, Christov, and Konstantinov's algorithm (hereafter, PCK) [10] for PAPSIS.

We assume the system to be initially in unreduced controller Hessenberg form [13]. This reduction can be carried out by means of efficient blocked algorithms based on (rank-revealing) orthogonal factorizations [12].

Our algorithms are specially designed to provide a better use of the cache memory, while maintaining the same numerical properties. The experimental results on SGI PowerChallenge and SUN Enterprise multiprocessors report the performance of our block-partitioned serial and parallel algorithms.

2 The sequential PCK algorithm

Consider the controllable single-input system in controller Hessenberg form defined by (A, b), with real entries,

$$
(b|A) = \begin{bmatrix} \beta_1 & \alpha_{11} & \ldots & \alpha_{1,n-1} & \alpha_{1n} \\ & \alpha_{21} & \ldots & \alpha_{2,n-1} & \alpha_{2n} \\ & & \ddots & \vdots & \vdots \\ & & & \alpha_{n,n-1} & \alpha_{nn} \end{bmatrix}. \tag{1}
$$

As the system is controllable, it can be shown that $\beta_1, \alpha_{21}, \ldots, \alpha_{n,n-1} \neq 0$ [13].

The PCK algorithm is based on orthogonal transformations of the eigenvectors and proceeds as follows. (For simplicity we only describe the algorithm for pole assignment of real eigenvalues.) Let $\lambda \in \mathbb{R}$ and $v \in \mathbb{R}^n$ be, respectively, an eigenvalue and its corresponding eigenvector of the closed-loop matrix $A - bf$. Let Q be an orthogonal matrix such that $Qv = (v_1, 0, \ldots, 0)^T$. This matrix can be constructed so that $Q^T AQ$ and $Q^T(A - bf)Q$ are in Hessenberg form. Furthermore,

$$
Q^T(A - bf)Qe_1 = (\lambda, 0, \ldots, 0)^T, \tag{2}
$$

where e_1 is the first column of the identity matrix, and solving (2) we find the first element of the transformed feedback $\bar{f} = fQ$ from the corresponding elements of $Q^T A Q$ and $Q^T b$. After this stage, the procedure is repeated with the lower trailing blocks of order $n-1$ of the transformed matrices to assign a new pole. By proceeding recursively we obtain $\bar{f}$, and $f = \bar{f}Q^T$. The procedure for assigning $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ can be roughly stated as follows.

```
for  i = 1,...,n - 1
    Set v_n = 1 and compute v_{n-1}
    for  j = n - 1, n - 2,...,i
        Compute v_{j-1}
        Construct a Givens rotation R_{i,j+1} ∈ ℝ^{n×n} such that
            (v_1,...,v_j,v_{j+1},0,...,0)R_{i,j+1} = (v_1,...,ṽ_j,0,...,0)
        Apply the transformation A = R_{i,j+1}AR^T_{i,j+1}
    end for
    Apply the transformation b = R_{i,i+1}b
    Compute f̄_i = a_{i+1,i}/b_{i+1}
end for
Compute  f̄_n = (a_{n,n} - λ_n)/b_n
```

At each iteration of the outer loop a new pole is assigned. In the inner loop, at each iteration we compute a component of eigenvector v $(j-1)$, obtain a transformation to introduce a zero in a component of the eigenvector $(j+1)$, and finally apply this transformation on the system matrix.

3   Parallelization of the PCK algorithm

In traditional implementations of this algorithm each transformation matrix $R_{i,j+1}$ is applied immediately after it is computed. Thus, at each iteration of loop j, two rows and columns (j-th and $j+1$-th) of the matrix are referenced.

Our block-partitioned algorithms reduce the number of data references by delaying the update of some entries the matrix. Thus, we work on the transformed lower Hessenberg matrix A^T, partition this matrix by blocks of columns (see figure 1), and delay the application of transformations from the left until the proper block is referenced. Although the parameters of the delayed transformations need to be stored, the dimension of this work space is small.

Specifically, consider the assignment of the first pole in the block-partitioned algorithm:

- A set of transformations are computed to shift up the pole, until it dissapears on the top left corner of block B1, and the transformations are only applied to B1. The application of this update from the left to blocks B2, ..., B6 is delayed.
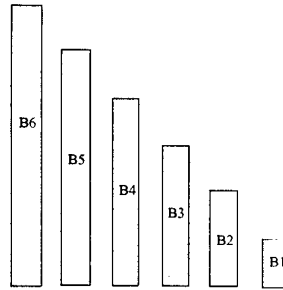
Fig. 1. Partition of the matrix by blocks of columns.

- The procedure continues with block B2. First, the delayed update is applied from the left to B2. Then, a new set of transformations are computed to shift up the pole, and these transformations are only applied to B2. (The application of this update from the left to blocks B3, ..., B6 is delayed.)
- The procedure is repeated with blocks B3, B4, B5, and B6, until the pole is assigned and the problem is deflacted.

In the parallel algorithm we are interested in an algorithm with a higher (and coarser) degree of parallelism than that achieved with the application of a single tranformation. Notice that in each iteration of the inner loop j two rows and two columns of the matrix are modified. Thus, as soon as $j = n - 4$, it would be possible to start the assignment of a different pole.

This is a pipelined algorithm. Specifically, the assignment of a new pole can be started as soon as the transformations related to the previous pole do not affect to the last block of columns. Thus, it is possible to assign in parallel as many poles as blocks in the partition of A^T.

In our algorithm, the maximum number of pipelined stages is $\frac{n/nb}{2}$, where n and nb are the problem size and block size respectively. Figure 2 shows the evolution of the different stages in our pipelined algorithm. As the problem is deflacted, the number of blocks of columns (and therefore the number of pipelined stages) decreases. In practice, nb must be larger than three; otherwise, the stages can not be correctly pipelined.

4   Experimental Results

In this section we report the results of our numerical experiments on a SGI PowerChallenge (SGI MIPS R10000) and a SUN Entreprise 4000 (SUN UltraSPARC) multiprocessors. All our experiments were performed using IEEE

(a) *Stage 1, step 1.*

(b) *Stage 1, step 2.*

(c) *Stage 1, step 3.*
   Stage 2, step 1.

(d) *Stage 1, step 4.*
   Stage 2, step 2.

(e) *Stage 1, step 5.*
   Stage 2, step 3.
   Stage 3, step 1.

(f) *Stage 1, step 6.*
   Stage 2, step 4.
   Stage 3, step 2.
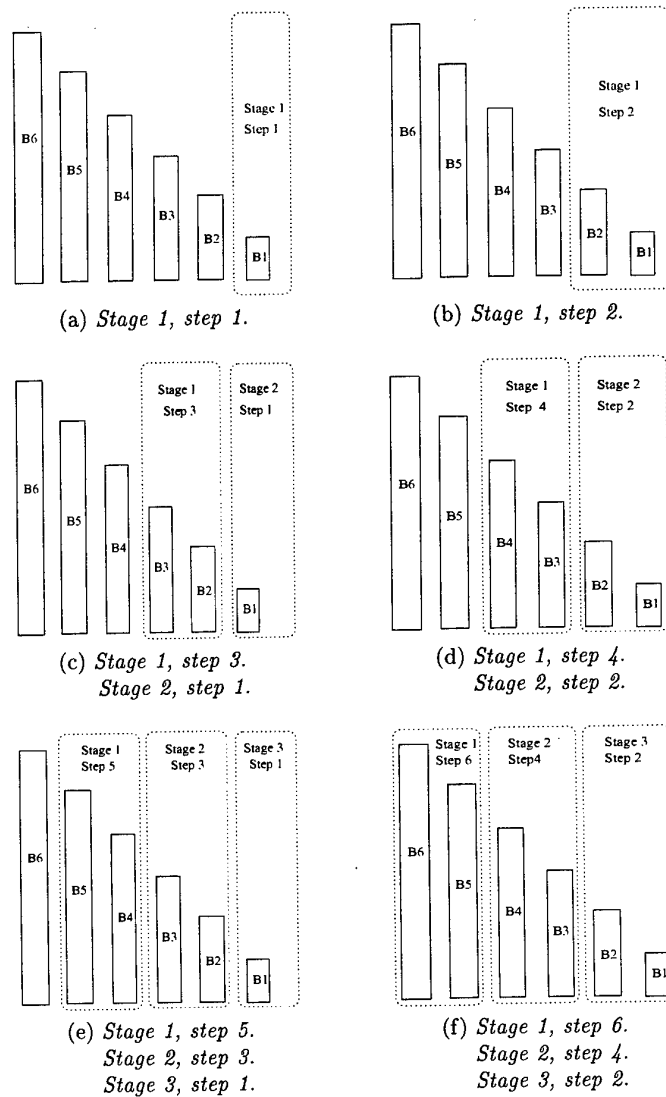
Fig. 2. Evolution of the pipelined algorithm.

double-precision arithmetic and Fortran-77 ($\epsilon \approx 2,2 \times 10^{-16}$). We have employed in our implementations orthogonal transformations based on Givens rotations. The system pair (A, b) was generated so that the computation of the feedback matrix was well-conditioned.

We have developed the following pole-assignment algorithms:

- BPAPSIS: Block-partitioned algorithm.
- PPAPSIS: Parallel version of the block-partitioned algorithm.

Figure 3 shows the speed-up of our block-partitioned algorithm for different block dimensions and problem sizes, nb and n respectively. We test system of moderate size from 100 to 1000, using block sizes of ($nb =$)1 (non-blocked algorithm), 32, 64 and 100 for the SGI MIPS R10000 processor, and $nb= 1, 16, 32$ and 64 for the SUN UltraSPARC processor. The results are averaged for 5 executions on different random matrices. In all the experiments the blocked implementations clearly outperform the sequential code ($nb = 1$), except on SGI MIPS R10000 when the problem size is reduced ($n \leq 200$).
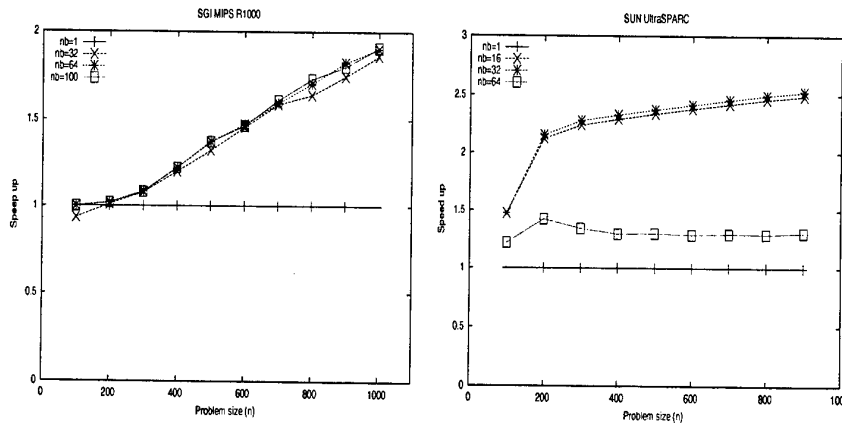
Fig. 3. Speed-up of the block-partitioned algorithm on the SGI MIPS R10000 (left) and the SUN UltraSPARC (right) processors.

Figure 4 shows the efficiency of our parallel algorithm compared with the non-blocked and blocked algorithms using $np = 2, 4, \ldots, 12$ processors. These figures report the efficiency versus problem size on the SGI PowerChallenge and SUN Enterprise platforms. The blocked and parallel algorithm employ the optimal block size determined in the previous experiment, i.e., $nb = 100$ and $nb = 32$ for SGI and SUN, respectively. As these figures show if

we compare our parallel algorithms with the serial algorithm (non-blocked) efficiencies higher than 1 are obtained. On the other hand if the paralell algorithm is compared with blocked algorithm, the maximum efficiency is 80% and decrease as the number of processors of the system is increased, since the problem size is moderate.
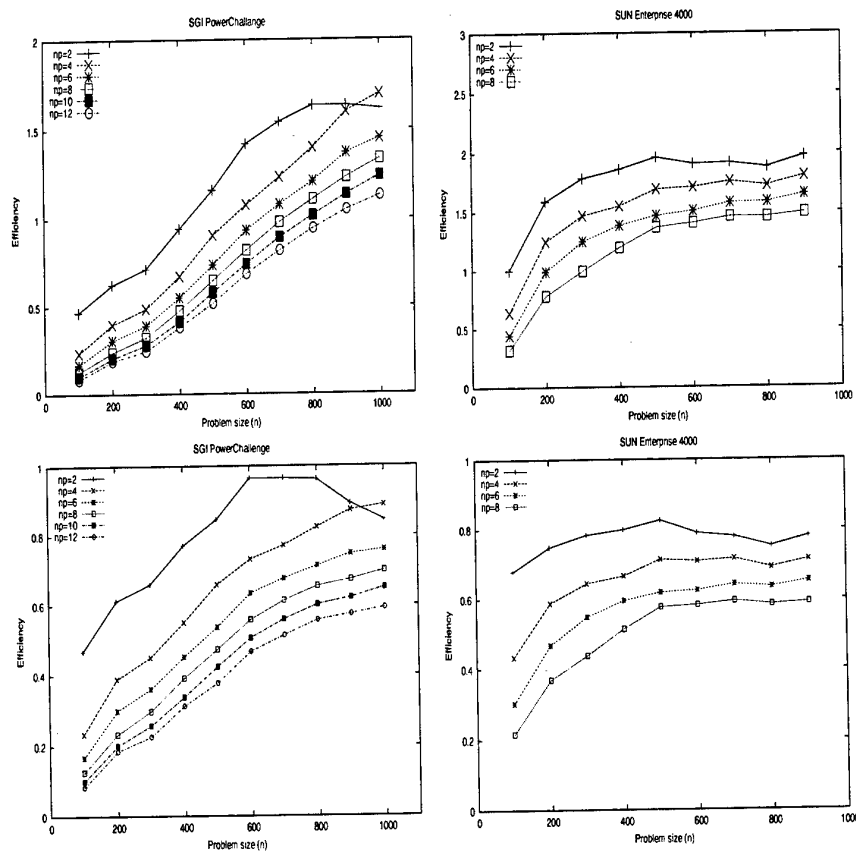
Fig. 4. Efficiency of the parallel algorithm compared with the non-blocked algorithm (top) and the blocked algorithm (bottom) on the SGI PowerChallenge (left) and the SUN Enterprise 4000 (right) multiprocessors.

5 Conclusions

We have presented block-partitioned and parallel versions of Petkov, Christov, and Konstantinov's algorithm for the pole assignment problem of single-input systems. Our block-partitioned algorithms achieve a high speed-up on SGI and SUN processors, while maintaining the same numerical properties.

The experimental results of the parallel algorithms also show an important increase in performance on an SGI PowerChallenge and SUN Enterprise platforms.

References

1. E. Anderson et al. *LAPACK User's Guide Release 2.0*. (SIAM, Philadelphia (USA), 1994).
2. M. Castillo, G. Quintana-Ortí, V. Hernández, E. S. Quintana-Ortí. *Block-partitioned algorithms for the pole assignment problem of single-input systems*. Proc. of the IFAC Conference on Systems Structure and Control - SSC'98, pp. 409–414, Nantes (France), July 1998.
3. C. L. Cox, W. F. Moss. *Backward error analysis for a pole assignment algorithm*. SIAM J. Matrix Analysis and Appl., Vol. 10, pp. 446–456, 1989.
4. T. Kailath. *Linear systems*. (Prentice-Hall, Englewood Cliffs (USA), 1980).
5. V. Mehrmann, H. Xu. *An analysis of the pole placement problem: I. The single-input case*. Elec. Trans. on Numer. Anal., Vol. 4, pp. 89–105, 1996.
6. G. Miminis, C. C. Paige. *An algorithm for pole assignment of time invariant linear systems*. Int. J. of Control, Vol. 35, pp. 341–354, 1982.
7. G. Miminis, C. C. Paige. *A direct algorithm for pole assignment of time-invariant multi-input systems using state feedback*. Automatica, Vol. 24, pp. 343–356, 1988.
8. G. Miminis, C. C. Paige. *Implicit shifting in the QR and related algorithms*. SIAM J. Matrix Analysis and Appl., Vol. 12, pp. 385–400, 1991.
9. R. V. Patel, P. Misra. *Numerical algorithm for eigenvalue assignment by state feedback*. Proc. IEEE, Vol. 72, pp. 1755–1764, 1984.
10. P. Hr. Petkov, N. D. Christov, M. M. Konstantinov. *A computational algorithm for pole assignment of linear single-input systems*. IEEE Trans. Autom. Control, AC-29, pp. 1045–1048, 1984.
11. P. Hr. Petkov, N. D. Christov, M. M. Konstantinov. *Computational methods for linear control systems*. (Prentice-Hall, Englewood Cliffs (USA), 1991).
12. G. Quintana-Ortí, X. Sun, C. H. Bischof. *A BLAS-3 version of the QR factorizaton with column pivoting*. SIAM J. Scientific Comp., Vol. 19, No. 5, pp. 1486–1494, 1998.
13. P. Van Dooren. *The generalized eigenstructure problem in linear systems theory*. IEEE Trans. Autom. Control, AC-31, pp. 1755–1764, 1981.
14. A. Varga. *A multishift Hessenberg algorithm for pole assignment of single-input systems*. IEEE Trans. Autom. Control, AC-41, pp. 1795–1799, 1996.
15. W. M. Wonham. *On pole assignment in multi-input controllable linear systems*. IEEE Trans. Autom. Control, AC-12, pp. 660–665, 1967.

Non-stationary Parallel Newton Iterative Methods for Nonlinear Problems*

Josep Arnal, Violeta Migallón, and José Penadés

Departamento de Ciencia de la Computación e Inteligencia Artificial,
Universidad de Alicante.
E-03071 Alicante, Spain
{arnal, violeta, jpenades}@dccia.ua.es

Abstract. Parallel algorithms for solving nonlinear systems are studied. Non-stationary parallel algorithms based on the Newton method are considered. Convergence properties of these methods are studied when the matrix in question is either monotone or an H-matrix. In order to illustrate the behavior of these methods, we implemented these algorithms on two distributed memory multiprocessors. The first platform is an Ethernet network of five 120 MHz Pentiums. The second platform is an IBM RS/6000 with 8 nodes. Several versions of these algorithms are tested. Experiments show that these algorithms can solve the nonlinear system in substantially less time that the current (stationary or non-stationary) parallel nonlinear algorithms based on the multisplitting technique.

Topics. Numerical methods, parallel and distributed algorithms.

1 Introduction

Let $F : \mathbb{R}^n \to \mathbb{R}^n$ be a nonlinear function. We are interested in the parallel solution of the system of nonlinear equations

$$F(x) = 0, \tag{1}$$

where it is assumed that a solution $x^\star$ exists. We suppose that there exists an $r_0 > 0$ such that

    (i) F is differentiable on $S_0 \equiv \{x \in \mathbb{R}^n \; : \; \|x - x^\star\| < r_0\}$,

    (ii) the Jacobian matrix at $x^\star$, $F'(x^\star)$, is nonsingular,

    (iii) there exists an $L > 0$ such that for $x \in S_0$, $\|F'(x) - F'(x^\star)\| \leq L\|x - x^\star\|$.

Under assumptions (i)-(iii), a well-known method for solving the nonlinear system (1) is the classical Newton method (cf. [11]). Given an initial vector $x^{(0)}$, this method produces the following sequence of vectors

$$x^{(\ell+1)} = x^{(\ell)} - x^{(\ell+\frac{1}{2})}, \quad \ell = 0, 1, \dots, \tag{2}$$

where $x^{(\ell+\frac{1}{2})}$ is the solution of the linear system

$$F'(x^{(\ell)})z = F(x^{(\ell)}). \tag{3}$$

On the other hand, if we use an iterative method to approximate the solution of (3) we are in the presence of a Newton iterative method; see e.g., [11] and [12]. In order to generate efficient algorithms to solve nonlinear system (1) on a parallel computer, White [14] defines the parallel Newton-SOR method, that generalizes a particular Newton iterative method, the Newton-SOR method. In [14], White also introduces a parallel nonlinear Gauss-Seidel algorithm for approximating the solution of an almost linear system, that is, to solve (1) when $F(x) = Ax + \Phi(x) - b$, where $A = (a_{ij})$ is a real $n \times n$ matrix, x and b are n-vectors and $\Phi : \mathbb{R}^n \to \mathbb{R}^n$ is a nonlinear diagonal mapping (i.e., the ith component Φ_i of Φ is a function only of x_i). Bai [2], has generalized the parallel nonlinear Gauss-Seidel algorithm in the context of relaxed methods. Both methods are based on the use of the multisplitting technique (see [10]). On the other hand, Bru, Elsner and Neumann [4] studied two non-stationary methods (synchronous and asynchronous) based on the multisplitting method for solving linear systems in parallel. As it can be seen e.g., in [6] and [9], non-stationary algorithms behave better than the multisplitting method. Recently, in [1] we have extended the idea of the non-stationary methods to the problem of solving an almost linear system. These methods are a generalization of the parallel nonlinear Gauss-Seidel algorithm [14] and the parallel nonlinear AOR method [2].

In this paper we construct a parallel Newton iterative algorithm to solve the general nonlinear system (1) that uses non-stationary multisplitting models to approximate linear system (3). For this purpose, let us consider for each x, a multisplitting of $F'(x)$, $\{M_k(x), N_k(x), E_k\}_{k=1}^p$, that is, a collection of splittings

$$F'(x) = M_k(x) - N_k(x), \quad 1 \le k \le p, \tag{4}$$

and diagonal nonnegative weighting matrices E_k which add to the identity. Let us further consider a sequence of integers $q(\ell, s, k)$, $\ell = 0, 1, 2, \ldots$, $s = 1, 2, \ldots, m_\ell$, $1 \le k \le p$, called non-stationary parameters. Following [4] or [9] the linear system (3) can be approximated by $x^{(\ell+\frac{1}{2})}$ as follows

$$x^{(\ell+\frac{1}{2})} = z^{(m_\ell)},$$
$$z^{(s)} = H_{\ell,s}(x^{(\ell)})z^{(s-1)} + B_{\ell,s}(x^{(\ell)})F(x^{(\ell)}), \quad s = 1, 2, \ldots, m_\ell,$$

$$H_{\ell,s}(x) = \sum_{k=1}^p E_k \left(M_k^{-1}(x)N_k(x) \right)^{q(\ell,s,k)}, \tag{5}$$

$$B_{\ell,s}(x) = \sum_{k=1}^p E_k \sum_{j=0}^{q(\ell,s,k)-1} \left(M_k^{-1}(x)N_k(x) \right)^j M_k^{-1}(x) \tag{6}$$

$$= \sum_{k=1}^p E_k \left(I - (M_k^{-1}(x)N_k(x))^{q(\ell,s,k)} \right) (F'(x))^{-1}, \tag{7}$$

and $z^{(0)} = 0$. Thus

$$x^{(\ell+\frac{1}{2})} = \left(\sum_{i=1}^{m_\ell-1} \prod_{j=i+1}^{m_\ell} H_{\ell,j}(x^{(\ell)}) B_{\ell,i}(x^{(\ell)}) + B_{\ell,m_\ell}(x^{(\ell)}) \right) F(x^{(\ell)}),$$

where $\prod_{j=i+1}^{m_\ell} H_{\ell,j}(x^{(\ell)})$ denotes the product of the matrices $H_{\ell,j}(x^{(\ell)})$ in the order $H_{\ell,m_\ell}(x^{(\ell)}) H_{\ell,m_\ell-1}(x^{(\ell)}) \ldots H_{\ell,i+1}(x^{(\ell)})$. Therefore, from (2) the non-stationary parallel Newton iterative method can be written as follows.

$$x^{(\ell+1)} = G_{\ell,m_\ell}(x^{(\ell)}), \tag{8}$$

where

$$G_{\ell,m_\ell}(x) = x - A_{\ell,m_\ell}(x)F(x)$$

and

$$A_{\ell,m_\ell}(x) = \left(\sum_{i=1}^{m_\ell-1} \prod_{j=i+1}^{m_\ell} H_{\ell,j}(x) B_{\ell,i}(x) + B_{\ell,m_\ell}(x) \right). \tag{9}$$

We note that the formulation of this method allows us to use different number of local iterations $q(\ell, s, k)$ not only in each processor k and at each nonlinear iteration ℓ but at each linear iteration s. Moreover, this method extends the parallel Newton method introduced by White [14].

In the following section we analyze the convergence properties of this algorithm when the Jacobian matrix is monotone or an H-matrix. Section 3 contains some numerical experiments, which illustrate the performance of the algorithms studied, on an Ethernet network of five 120 MHz Pentiums and on an IBM RS/6000 SP. In the rest of this section we present some notation, definitions and preliminary results used in the paper.

A matrix A is said to be a nonsingular M-matrix if A has all nonpositive off-diagonal entries and it is monotone, i.e., $A^{-1} \geq O$. For any matrix $A = (a_{ij}) \in \mathbb{R}^{n \times n}$, we define its comparison matrix $\langle A \rangle = (\alpha_{ij})$ by $\alpha_{ii} = |a_{ii}|$, $\alpha_{ij} = -|a_{ij}|$, $i \neq j$. The matrix A is said to be an H-matrix if $\langle A \rangle$ is a nonsingular M-matrix. The splitting $A = M - N$ is called a weak regular splitting if $M^{-1} \geq O$ and $M^{-1}N \geq O$; the splitting is an H-compatible splitting if $\langle A \rangle = \langle M \rangle - |N|$; see e.g., Berman and Plemmons [3] or Varga [13].

A sequence $\{x^{(\ell)}\}$ converges Q-quadratically to x^* if there exists $c < 1$ such that

$$\|x^{(\ell+1)} - x^*\| \leq c\|x^{(\ell)} - x^*\|^2.$$

$L(\mathbb{R}^n)$ denotes the linear space of linear operators from $\mathbb{R}^n$ to $\mathbb{R}^n$.

Lemma 1. *Suppose that the mapping* $A : D \subset \mathbb{R}^m \to L(\mathbb{R}^n)$ *is continuous at a point* $x^0 \in D$ *for which* $A(x^0)$ *is nonsingular. Then there is a* $\delta > 0$ *and a* $\beta > 0$ *so that, for any* $x \in D \cap \{x : \|x - x^0\| \leq \delta\}$, $A(x)$ *is nonsingular and* $\|A(x)^{-1}\| \leq \beta$. *Moreover,* $A(x)^{-1}$ *is continuous in* x *at* x^0.

Proof. See Ortega and Rheinboldt [11].

Theorem 1. *Suppose $F : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ is G-differentiable at each point of a convex set $D_0 \subseteq D$, then for any $x, y, z \in D_0$,*

$$\|F(y) - F(z) - F'(x)(y - z)\| \leq \|y - z\| \sup_{0 \leq t \leq 1} \|F'(z + t(y - z)) - F'(x)\|.$$

Proof. See Ortega and Rheinboldt [11].

2 Convergence

In this section we study the convergence of the iterative scheme (8). For this purpose we need to make the following additional assumptions on the splittings (4).

(iv) There exist $t_k > 0$, $1 \leq k \leq p$, such that for $x \in S_0$, $\|M_k(x) - M_k(x^\star)\| \leq t_k\|x - x^\star\|$.

(v) $M_k(x^\star)$, $1 \leq k \leq p$, are nonsingular.

(vi) There exits $0 \leq \alpha < 1$, such that, for each positive integer s and $\ell = 0, 1, \ldots$,

$$\|H_{\ell,s}(x^\star)\| \leq \alpha,$$

where $H_{\ell,s}(x^\star)$ is defined in (5).

From assumptions (i)-(iii) of Section 1 and using Lemma 1, it follows that there exists $0 < r_1 \leq r_0$ such that F' is continuous and nonsingular in $S_1 = \{x \in \mathbb{R}^n : \|x - x^\star\| < r_1\}$. On the other hand, it can be shown (see e.g., Ortega and Rheinboldt [11]) that Newton method (2) converges Q-quadratically to $x^\star$ in a neighborhood of $x^\star$. In order to simplify the notation we also denote this neighborhood by S_1. From assumptions (iv)-(v) and Lemma 1, it follows that M_k, $1 \leq k \leq p$, is continuous and nonsingular in a neighborhood of $x^\star$, say again S_1. Therefore $M_k(x)^{-1}N_k(x)$, $1 \leq k \leq p$, is well defined and moreover continuous in S_1. Then, $H_{\ell,s}(x)$ is also continuous in S_1. Now, from assumption (vi) it obtains that $\|H_{\ell,s}(x)\| \leq \alpha$, $\ell = 0, 1, \ldots$, $s = 1, 2, \ldots, m_\ell$, in a neighborhood of $x^\star$, denoted again by S_1. Moreover, since $M_k(x)^{-1}N_k(x)$, $1 \leq k \leq p$, are continuous in S_1, there exists a positive integer K such that

$$\|M_k(x)^{-1}N_k(x)\| \leq K, \quad 1 \leq k \leq p, \tag{10}$$

for all x in a neighborhood of $x^\star$, that we denote again by S_1.

Lemma 2. *Let $A : \mathbb{R}^n \to L(\mathbb{R}^n)$ be a mapping such that $\|A(x)\| \leq \delta$, in a neighborhood S of $x^\star$. Then for any $x \in S$ and for any positive integer m*

$$\|A(x)^m - A(x^\star)^m\| \leq m\delta^{m-1}\|A(x) - A(x^\star)\|.$$

Proof. We proceed by induction. For $m = 1$, the result follows obviously. Suppose that the result is true for $m = k$. Then

$$\| A(x)^{k+1} - A(x^\star)^{k+1}\| = \|A(x)^{k+1} - A(x)^k A(x^\star) + A(x)^k A(x^\star) - A(x^\star)^{k+1}\|$$
$$\leq \|A(x)^k(A(x) - A(x^\star))\| + \|(A(x)^k - A(x^\star)^k)A(x^\star)\|.$$
$$\leq \delta^k\|A(x) - A(x^\star)\| + k\delta^{k-1}\|A(x) - A(x^\star)\|\delta = (k+1)\delta^k\|A(x) - A(x^\star)\|,$$

and the proof is complete.

Lemma 3. *Let $x \in S_1$ and let m be a positive integer, then*

$$I - \prod_{j=1}^{m} H_{\ell,j}(x) = A_{\ell,m}(x) F'(x) \qquad \ell = 0, 1, \dots.$$

Proof. Let $x \in S_1$, then from (7)

$$B_{\ell,s}(x) F'(x) = I - H_{\ell,s}(x), \quad s = 1, 2, \dots, m, \ \ell = 0, 1, \dots, \qquad (11)$$

where $H_{\ell,s}(x)$ and $B_{\ell,s}(x)$ are defined in (5) and (6) respectively. Then from (9) and (11) we obtain

$$\begin{aligned}
A_{\ell,m}(x) F'(x) &= \sum_{i=1}^{m-1} \prod_{j=i+1}^{m} H_{\ell,j}(x) \left(I - H_{\ell,i}(x)\right) + \left(I - H_{\ell,m}(x)\right) \\
&= \sum_{i=1}^{m-1} \left(\prod_{j=i+1}^{m} H_{\ell,j}(x) - \prod_{j=i}^{m} H_{\ell,j}(x) \right) + \left(I - H_{\ell,m}(x)\right) \\
&= I - \prod_{j=1}^{m} H_{\ell,j}(x),
\end{aligned}$$

and the proof is done.

Lemma 4. *Let $x \in S_1$ and let m be a positive integer, then*

$$\| \prod_{j=1}^{m} H_{\ell,j}(x) - \prod_{j=1}^{m} H_{\ell,j}(x^\star) \| \leq \alpha^{m-1} \sum_{j=1}^{m} \| H_{\ell,j}(x) - H_{\ell,j}(x^\star) \|, \quad \ell = 0, 1, \dots.$$

Proof. In order to show this result, we proceed by induction. Obviously, the result follows for $m = 1$. Suppose that the result is true for $m = k$. Then taking into account that $\| H_{\ell,s}(x) \| \leq \alpha$, we can write

$$\begin{aligned}
\| \prod_{j=1}^{k+1} H_{\ell,j}(x) - \prod_{j=1}^{k+1} H_{\ell,j}(x^\star) \| &= \| \prod_{j=1}^{k+1} H_{\ell,j}(x) - H_{\ell,k+1}(x) \prod_{j=1}^{k} H_{\ell,j}(x^\star) \\
&\quad + H_{\ell,k+1}(x) \prod_{j=1}^{k} H_{\ell,j}(x^\star) - \prod_{j=1}^{k+1} H_{\ell,j}(x^\star) \| \\
&\leq \| H_{\ell,k+1}(x) \| \| \prod_{j=1}^{k} H_{\ell,j}(x) - \prod_{j=1}^{k} H_{\ell,j}(x^\star) \| \\
&\quad + \| H_{\ell,k+1}(x) - H_{\ell,k+1}(x^\star) \| \| \prod_{j=1}^{k} H_{\ell,j}(x^\star) \| \\
&\leq \alpha \| \prod_{j=1}^{k} H_{\ell,j}(x) - \prod_{j=1}^{k} H_{\ell,j}(x^\star) \| + \| H_{\ell,k+1}(x) - H_{\ell,k+1}(x^\star) \| \alpha^k
\end{aligned}$$

$$\leq \alpha\alpha^{k-1} \sum_{j=1}^{k} \|H_{\ell,j}(x) - H_{\ell,j}(x^*)\| + \alpha^k \|H_{\ell,k+1}(x) - H_{\ell,k+1}(x^*)\|$$

$$= \alpha^k \sum_{j=1}^{k+1} \|H_{\ell,j}(x) - H_{\ell,j}(x^*)\|,$$

and the proof is complete.

Lemma 5. *Suppose assumptions (i)-(v) are satisfied. Assume further that the sequence of number of local iterations $q(\ell, s, k)$, $\ell = 0, 1, \ldots$, $s = 1, 2, \ldots, m_\ell$, $1 \leq k \leq p$, remains bounded by $q > 0$. Then, there exists $L^* > 0$ such that, for any $x \in S_1$ and for any positive integer s, it follows*

$$\|H_{\ell,s}(x) - H_{\ell,s}(x^*)\| \leq L^* \|x - x^*\|, \quad \ell = 0, 1, \ldots.$$

Proof. Let $x \in S_1$, from (iii), (iv) y (v) it is known (see e.g., [12]) that there exists $r_k > 0$, $1 \leq k \leq p$, such that

$$\|M_k^{-1}(x)N_k(x) - M_k^{-1}(x^*)N_k(x^*)\| \leq r_k \|x - x^*\|. \tag{12}$$

On the other hand, if we denote $R_k(x) = M_k^{-1}(x)N_k(x)$, using Lemma 2 and (10), it obtains

$$\|R_k(x)^{q(\ell,s,k)} - R_k(x^*)^{q(\ell,s,k)}\| \leq q(\ell,s,k)K^{q(\ell,s,k)-1}\|R_k(x) - R_k(x^*)\|. \tag{13}$$

Therefore from (10), (12) and (13), we have

$$\|H_{\ell,s}(x) - H_{\ell,s}(x^*)\| \leq \sum_{k=1}^{p} \|E_k\| \|R_k^{q(\ell,s,k)}(x) - R_k^{q(\ell,s,k)}(x^*)\|$$

$$\leq \sum_{k=1}^{p} \|E_k\| q(\ell,s,k)K^{q(\ell,s,k)-1}r_k\|x - x^*\| \leq \sum_{k=1}^{p} \|E_k\| \left(qK'^{q-1}r_k \right) \|x - x^*\|, \tag{14}$$

with $K' = \max\{1, K\}$. Then $\|H_{\ell,s}(x) - H_{\ell,s}(x^*)\| \leq L^* \|x - x^*\|$, with $L^* = \sum_{k=1}^{p} \|E_k\| \left(qK'^{q-1}r_k \right)$.

Lemma 6. *Let assumptions (i)-(vi) hold and suppose that the sequence of number of local iterations $q(\ell, s, k)$, $\ell = 0, 1, \ldots$, $s = 1, 2, \ldots, m_\ell$, $1 \leq k \leq p$, remains bounded by $q > 0$, then there exists $c_1 < +\infty$, such that for any $x \in S_1$,*

$$\|G_{\ell,m}(x) - x^*\| \leq c_1 \|x - x^*\|^2 + \alpha^m \|x - x^*\|, \quad \ell = 0, 1, \ldots,$$

Proof. From Lemma 3 it follows

$$\|G_{\ell,m}(x) - x^*\| = \|x - A_{\ell,m}(x)F(x) - x^*\|$$

$$\leq \| - A_{\ell,m}(x)F(x) + (I - \prod_{j=1}^{m} H_{\ell,j}(x^*))(x - x^*)\| + \|\prod_{j=1}^{m} H_{\ell,j}(x^*)(x - x^*)\|$$

$$= \| - A_{\ell,m}(x)F(x) + A_{\ell,m}(x^*)F'(x^*)(x - x^*)\| + \|\prod_{j=1}^{m} H_{\ell,j}(x^*)(x - x^*)\|.$$

Then by assumption (vi), for $\ell = 0, 1, \ldots$, it obtains

$$\|G_{\ell,m}(x) - x^\star\| \leq \| - A_{\ell,m}(x)F(x) + A_{\ell,m}(x^\star)F'(x^\star)(x - x^\star)\| + \alpha^m \|x - x^\star\|.$$

Now, since $F(x^\star) = 0$, we have the following inequalities

$$\begin{aligned}
\|G_{\ell,m}(x) - x^\star\| \leq & \| - A_{\ell,m}(x)\left(F(x) - F(x^\star) - F'(x^\star)(x - x^\star)\right)\| \\
& + \|A_{\ell,m}(x)\left(F'(x) - F'(x^\star)\right)(x - x^\star)\| \\
& + \|\left(A_{\ell,m}(x)F'(x) - A_{\ell,m}(x^\star)F'(x^\star)\right)(x - x^\star)\| \\
& + \alpha^m \|x - x^\star\|.
\end{aligned} \tag{15}$$

On the other hand from (9), and using assumption (vi) we have

$$\begin{aligned}
\|A_{\ell,m}(x)\| &= \left\| \sum_{i=1}^{m-1} \prod_{j=i+1}^{m} H_{\ell,j}(x)B_{\ell,i}(x) + B_{\ell,m}(x) \right\| \\
&\leq \sum_{i=1}^{m-1} \| \prod_{j=i+1}^{m} H_{\ell,j}(x)\|\|B_{\ell,i}(x)\| + \|B_{\ell,m}(x)\| \\
&\leq \sum_{i=1}^{m-1} \alpha^{m-i}\|B_{\ell,i}(x)\| + \|B_{\ell,m}(x)\|.
\end{aligned} \tag{16}$$

By the definition of $B_{\ell,s}(x)$, given by (6), and using (10), it obtains

$$\begin{aligned}
\|B_{\ell,s}(x)\| &\leq \sum_{k=1}^{p} \|E_k\| \sum_{h=0}^{q(\ell,s,k)-1} \|(M_k^{-1}(x)N_k(x))^h(x)\|\|M_k^{-1}(x)\| \\
&\leq \sum_{k=1}^{p} \|E_k\| \sum_{h=0}^{q(\ell,s,k)-1} K^h\|\|M_k^{-1}(x)\|.
\end{aligned}$$

That is, since the sequence $q(\ell, s, k)$, $\ell = 0, 1, \ldots$, $s = 1, 2, \ldots, m_\ell$, $1 \leq k \leq p$, remains bounded by $q > 0$, we have

$$\|B_{\ell,s}(x)\| \leq \sum_{k=1}^{p} \|E_k\| \sum_{h=0}^{q-1} K^h\|\|M_k^{-1}(x)\|. \tag{17}$$

Let $\beta = \max\{\beta_1, \beta_2, \ldots, \beta_p\}$, where $\beta_k = \sup\{\|M_k(x)^{-1}\| : x \in S_1\}$. The existence of β_k, $1 \leq k \leq p$, follows from Lemma 1. Then, from (17) it obtains

$$\|B_{\ell,s}(x)\| \leq \sum_{k=1}^{p} \|E_k\| \sum_{h=0}^{q-1} K^h \beta.$$

Thus,

$$\|B_{\ell,s}(x)\| \leq \bar{K}^\star, \quad \ell = 0, 1, \ldots, \quad s = 1, 2, \ldots, m_\ell, \tag{18}$$

for all $x \in S_1$, where $\bar{K}^\star = \sum_{k=1}^{p} \|E_k\| \sum_{h=0}^{q-1} K^h \beta > 0$.

Now, by (16) and (18), for any $x \in S_1$ and for any positive integer m, we have

$$\|A_{\ell,m}(x)\| \leq \sum_{i=1}^{m-1} \alpha^{m-i}\bar{K}^\star + \bar{K}^\star < \left(\frac{1}{1-\alpha}+1\right)\bar{K}^\star \equiv K^\star \qquad (19)$$

Now, using (19) we bound (15). From Theorem 1, it follows

$$\| - A_{\ell,m}(x)(F(x) - F(x^\star) - F'(x^\star)(x - x^\star))\| \leq$$

$$\leq \| - A_{\ell,m}(x)\|\|(x - x^\star)\| \sup_{0 \leq t \leq 1} \|(F(x^\star + t(x - x^\star)) - F'(x^\star))\|$$

$$\leq \| - A_{\ell,m}(x)\|\|(x - x^\star)\| \sup_{0 \leq t \leq 1} L\|t(x - x^\star)\| \leq K^\star L\|(x - x^\star)\|^2.$$

On the other hand, by condition (iii) we have

$$\|A_{\ell,m}(x)\left(F'(x) - F'(x^\star)\right)(x - x^\star)\| \leq \|A_{\ell,m}(x)\|\|F'(x) - F'(x^\star)\|\|(x - x^\star)\|$$
$$\leq K^\star L\|(x - x^\star)\|^2.$$

Using Lemmata 3, 4 and 5 it obtains

$$\|(A_{\ell,m}(x)F'(x) - A_{\ell,m}(x^\star)F'(x^\star))(x - x^\star)\| =$$

$$= \left\|\left(\prod_{j=1}^{m} H_{\ell,j}(x) - \prod_{j=1}^{m} H_{\ell,j}(x^\star)\right)(x - x^\star)\right\|$$

$$\leq \alpha^{m-1}\sum_{j=1}^{m}\|H_{\ell,j}(x) - H_{\ell,j}(x^\star)\|\|x - x^\star\| \leq m\alpha^{m-1}L^\star\|x - x^\star\|^2.$$

Since $\alpha < 1$, $\{m\alpha^{m-1}\}$ is upper bounded. Let c_2 (dependent of α) an upper bound of this set, then setting $c_1 = 2K^\star L + c_2 L^\star$, the proof is complete.

Remark 1. We want to point out that since we know nothing about the bound K in (10), we need, in lemmata 5 and 6, the sequence $q(\ell, s, k)$ to be bounded by $q > 0$. If we have $K < 1$, then we do not need that upper bound for the non-stationary parameters $q(\ell, s, k)$ (see (14) and (17)).

Theorem 2. *Let assumptions (i) (vi) hold and $F(x^\star) = 0$. Let $\{m_\ell\}_{\ell=0}^{\infty}$ be a sequence of positive integers, and define*

$$m = \max\left[\{m_0\} \cup \left\{m_\ell - \sum_{i=0}^{\ell-1} m_i \ : \ \ell = 1, 2, \dots\right\}\right]. \qquad (20)$$

Suppose that $m < +\infty$ and that the sequence of non-stationary parameters $q(\ell, s, k)$, $\ell = 0, 1, \dots, s = 1, 2, \dots, m_\ell$, $1 \leq k \leq p$, is bounded by $q > 0$. Then,

there exist $r > 0$ and $c < 1$ such that, for $x^{(0)} \in S \equiv \{x \in \mathbb{R}^n : \|x - x^*\| < r\}$, the sequence of iterates defined by (8) converges to x^* and satisfies

$$\|x^{(\ell+1)} - x^*\| \leq c^{m_\ell}\|x^{(\ell)} - x^*\|.$$

Proof. Let c_1 be as in Lemma 6. Let $c > 0$ be such that $\alpha^{1/m} < c < 1$. Since $\alpha < c^m$, there exists $0 < r < r_1$, such that

$$c_1 r + \alpha \leq c^m,$$

and then,

$$(c_1 r + \alpha)^{1/m} \leq c < 1.$$

Now, we proceed by induction. For $\ell = 1$, using Lemma 6, we have

$$\|x^{(1)} - x^*\| = \|G_{0,m_0}(x^{(0)}) - x^*\| \leq c_1\|x^{(0)} - x^*\|^2 + \alpha^{m_0}\|x^{(0)} - x^*\|$$
$$\leq (c_1 r + \alpha^{m_0})\|x^{(0)} - x^*\|.$$

Since $c_1 r + \alpha^{m_0} \leq c_1 r + \alpha \leq c^m \leq c^{m_0}$, then $\|x^{(1)} - x^*\| \leq c^{m_0}\|x^{(0)} - x^*\|$. Therefore the result follows for $\ell = 1$. Suppose that the result is true for $0 \leq \ell \leq j$. Then

$$\|x^{(j)} - x^*\| \leq c^{m_{j-1}}\|x^{(j-1)} - x^*\| \leq \prod_{s=0}^{j-1} c^{m_s}\|x^{(0)} - x^*\|.$$

Now, for $\ell = j + 1$, from Lemma 6 it follows

$$\|x^{(j+1)} - x^*\| = \|G_{j,m_j}(x^{(j)}) - x^*\| \leq (c_1\|x^{(j)} - x^*\| + \alpha^{m_j})\|x^{(j)} - x^*\|$$

$$\leq (c_1(\prod_{s=0}^{j-1} c^{m_s})\|x^{(0)} - x^*\| + \alpha^{m_j})\|x^{(j)} - x^*\|$$

$$\leq (c_1 r(\prod_{s=0}^{j-1} c^{m_s}) + \alpha^{m_j})\|x^{(j)} - x^*\| \leq (c_1 r c^{m_j - m} + \alpha^{m_j})\|x^{(j)} - x^*\|$$

$$\leq ((c^m - \alpha)c^{m_j - m} + \alpha^{m_j})\|x^{(j)} - x^*\|$$
$$= c^{m_j}((c^m - \alpha)c^{-m} + \alpha^{m_j}c^{-m_j})\|x^{(j)} - x^*\|$$
$$= c^{m_j}(1 - \alpha c^{-m} + \alpha^{m_j}c^{-m_j})\|x^{(j)} - x^*\|$$
$$= c^{m_j}(1 + \alpha c^{-m}(\alpha^{m_j - 1}c^{m - m_j} - 1))\|x^{(j)} - x^*\|.$$

Since $0 < \alpha < c^m < 1$, then $\alpha c^{-m} < 1$. On the other hand, $0 < \alpha^{m_j - 1}c^{m - m_j} \leq c^{m(m_j-1)}c^{m-m_j} = c^{m_j(m-1)} \leq 1$ and then, $-1 < \alpha c^{-m}(\alpha^{m_j - 1}c^{m - m_j} - 1) \leq 0$. Therefore, $\|x^{(j+1)} - x^*\| \leq c^{m_j}\|x^{(j)} - x^*\|$, and the proof is complete.

Theorem 3. *Let assumptions (i)-(iv) hold and $F(x^*) = 0$. Let $\{m_\ell\}_{\ell=0}^{\infty}$, be a sequence of positive integers, and define m as in (20). Suppose that $m < +\infty$. If any of the following two conditions is satisfied*
1. $F'(x^*)$ is a monotone matrix and $F'(x^*) = M_k(x^*) - N_k(x^*)$, $1 \leq k \leq p$, are

weak regular splittings,

2. $F'(x^\star)$ *is an H-matrix,* $F'(x^\star) = M_k(x^\star) - N_k(x^\star)$, $1 \leq k \leq p$, *are H-compatible splittings,*

then, there exist $r > 0$ and $c < 1$ such that, for $x^{(0)} \in S \equiv \{x \in \mathbb{R}^n \; : \; \|x - x^\star\| < r\}$, the sequence of iterates defined by (8) converges to $x^\star$ and satisfies

$$\|x^{(\ell+1)} - x^\star\| \leq c^{m_\ell}\|x^{(\ell)} - x^\star\|.$$

Proof. Under conditions 1 and 2 and taking into account respectively, the proofs of Theorem 2.1 of [4] and Theorem 3.1 of [9] we obtain that assumptions (v), (vi) and (10) with $K < 1$ are satisfied. Then, the proofs follow from Theorem 2 and Remark 1.

3 Numerical Experiments

We have implemented the above method on two distributed multiprocessors. The first platform is an IBM RS/6000 SP with 8 nodes. The second platform is an Ethernet network of five 120 MHz Pentiums. In order to manage the parallel environment we have used the PVMe library of parallel routines for the IBM RS/6000 SP and the PVM library for the cluster of Pentiums [7], [8].

In order to illustrate the behavior of the above algorithms, we have considered the following semilinear elliptic partial differential equation (see e.g., [5], [12], [14])

$$\begin{aligned}
-(K^1 u_x)_x - (K^2 u_y)_y &= -ge^u & (x,y) \in \Omega, \\
u &= x^2 + y^2 & (x,y) \in \partial\Omega,
\end{aligned} \tag{21}$$

where

$$\begin{aligned}
K^1 &= K^1(x,y) = 1 + x^2 + y^2, \\
K^2 &= K^2(x,y) = 1 + e^x + e^y, \\
g &= g(x,y) = 2(2 + 3x^2 + y^2 + e^x + (1+y)e^y)e^{-x^2-y^2}, \\
\Omega &= (0,1) \times (0,1).
\end{aligned}$$

It is well known that this problem has the unique solution $u(x,y) = x^2 + y^2$. To solve equation (21) using the finite difference method, we consider a grid in Ω of d^2 nodes equally spaced by $h = \Delta x = \Delta y = \frac{1}{d+1}$. This discretization yields a nonlinear system of the form $Ax + \Phi(x) = b$, where $\Phi : \mathbb{R}^n \to \mathbb{R}^n$ is a nonlinear diagonal mapping and A is a block tridiagonal symmetric matrix $A = (D_{i-1}, T_i, D_i)_{i=1}^d$, where T_i are tridiagonal matrices of size $d \times d$, $i = 1, 2, \ldots, d$, and D_i are $d \times d$ diagonal matrices, $i = 1, \ldots, d-1$; see e.g., [5]. Let $S = \{1, 2, \ldots, n\}$ and let S_k, $k = 1, 2, \ldots, p$, be subsets of S such that $S = \bigcup_{k=1}^p S_k$. Let us further consider a multisplitting of $F'(x)$, where $F(x) = Ax + \Phi(x) - b$, of the form

$$\{D(x) - L_k, U_k, E_k\}_{k=1}^p, \quad \text{where} \quad L_k \equiv \begin{cases} -a_{ij}, & j < i \text{ and } i, j \in S_k, \\ 0, & \text{otherwise}, \end{cases}$$

with

$$S_k = \{1 + \sum_{i<k} n_i, \ldots, \sum_{i=1}^{k} n_i\}, \quad 1 \le k \le p, \quad \sum_{k=1}^{p} n_k = n, \quad n_k > 0,$$

$$D(x) = \text{diag}(A) + \text{diag}(\Phi_1'(x_1), \ldots, \Phi_n'(x_n)),$$

and the $n \times n$ nonnegative diagonal matrices E_k, $1 \le k \le p$, are defined such that their ith diagonal entry is null if $i \notin S_k$. Note that this multisplitting is a Gauss-Seidel type multisplitting. The stopping criterion used was $\|x^{(\ell)} - v\|_2 \le h^2$, where $\| \cdot \|_2$ is the Euclidean norm and v is the vector which entries are the values of the exact solution of (21) on the nodes (ih, jh), $i, j = 1, \ldots, d$ and the initial vector was $x^{(0)} = (1, \ldots, 1)^T$. All times are reported in seconds.

We have run our codes with matrices of various sizes and different multi-splittings depending on the number of processors used (p) and the choice of the values n_k, $1 \le k \le p$, but to focus our discussion, we present here results obtained with $d = 64$, that originates a nonlinear system of size 4096. The conclusions we present here can be considered as representative of the larger set of experiments performed.

	$m_\ell = 1$	$m_\ell = 2$	$m_\ell = \ell$	$m_\ell = 2^\ell$
q=1	27.59	15.91	4.73	7.73
q=4	8.08	5.11	2.35	4.11
q=9	4.93	3.43	2.27	1.86
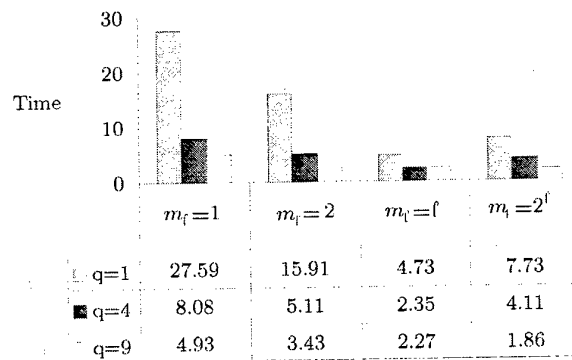
Fig. 1. Non-stationary parallel Newton Gauss-Seidel methods

Figure 1 shows the behavior of some non-stationary parallel Newton iterative methods on an IBM RS/6000 SP multiprocessor using four processors and $n_k = 1024$, $1 \le k \le 4$. This figure illustrates the influence of the non-stationary parameters $q(k) = q$, $1 \le k \le 4$, in relation to $m_\ell = 1$, 2, ℓ, 2^ℓ. We want to note that, for a fixed number of processors, the computational time starts to decrease as the non-stationary parameters increases until some optimal value of q ($q = 9$, in Figure 1) after which time starts to increase. This behavior is typical of non-stationary methods; see e.g. [6] and [9]. In general, this optimal value is hard to predict but if the decrease in the iterations balances the realization of more

local updates then less execution time is observed. This situation is independent of the choice of m_ℓ. On the other hand, in this figure it can also be observed that the best non-stationary parallel methods were obtained setting $m_\ell = \ell$ and $m_\ell = 2^\ell$.

	Cluster $m_1 = \ell$	IBM SP $m_1 = \ell$
q=1	28.33	5.36
q=2	18.99	3.93
q=10	14.01	3.01
q=16	14.54	3.16
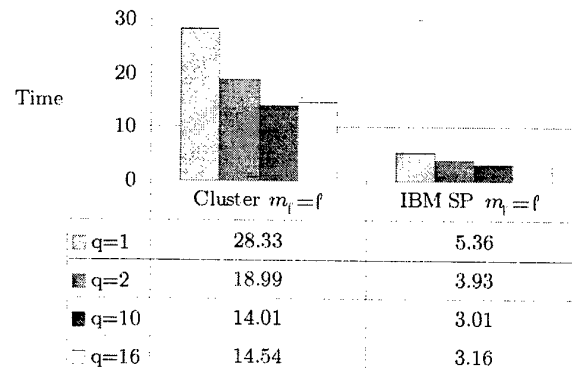
Fig. 2. Cluster of Pentiums versus IBM RS/6000 SP (2 processors)

Figure 2 shows the behavior of some non-stationary parallel Newton iterative methods in relation to the parallel computer system used. In this figure we have used two processors, $n_k = 2048$, $k = 1, 2$, and $m_\ell = \ell$. The conclusions were similar on both multiprocessors, however, the computing platform has obviously an influence in the performance of a parallel implementation. Note that when $q = 1$, the method reduces to the well-known parallel Newton Gauss-Seidel method (see [14]) and as it can be appreciated this method is always worse than the non-stationary parallel methods. Moreover, we have compared these methods with the algorithms presented in [1]. We have observed that the methods discussed here behave better than those algorithms. For example, for the matrix of size 4096, the best time we have obtained with the IBM RS/6000 SP using four processors (see Figure 1) is 1.86 seconds, however the best times obtained with the other methods (see Table 1 and 2 of [1]) were about 6 seconds.

On the other hand in Figure 3 we have compared the algorithms of this paper, setting $q = 9$, with the well-known sequential Newton Gauss-Seidel method [11] versus the number of processors in the IBM RS/6000 SP. The best CPU time performed by this sequential method was obtained with $m_\ell = \ell$. So, if we calculate the speed-up setting such sequential method as reference algorithm (i.e., $\dfrac{\text{CPU time of sequential Newton-Gauss Seidel algorithm, } (m_\ell = \ell)}{\text{REAL time of parallel algorithm}}$), it can be obtained an efficiency ($\dfrac{\text{Speed-up}}{\text{processors's number}}$) about 90% with two processors and about 60% with four processors. Similar efficiencies were obtained for the cluster of Pentiums. However it does not happen the same with the parallel New-

	$p = 2$	$p = 4$
$m_l = l$	3.1	2.27
$m_l = 2^l$	3.04	1.86
New-GS $m_l = l$	5.52	5.52
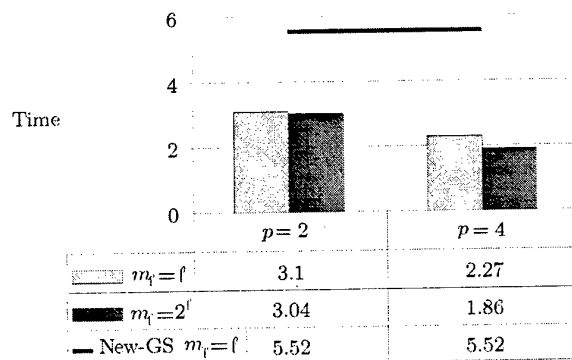
Fig. 3. Non-stationary methods $(q = 9)$ and sequential Newton Gauss-Seidel method

ton Gauss-Seidel method ([14]). That is, if $q = 1$, we have obtained efficiencies only about $0 - 30\%$ in both multiprocessors.
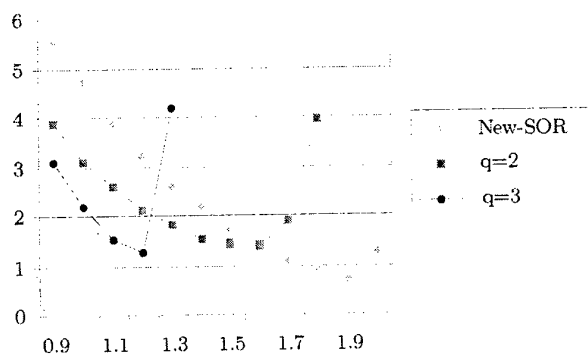
Fig. 4. Non-stationary Newton-SOR methods

Finally, Figure 4 illustrates the influence of the relaxation parameter ω when non-stationary parallel Newton-SOR methods are used. In this figure we have considered some non-stationary parallel Newton-SOR methods using four processors, $n_k = 1024$, $1 \leq k \leq 4$, and $m_\ell = \ell$, and for each one we recorded the REAL time in seconds on the IBM RS/6000 SP. Moreover, these results were compared to the corresponding parallel Newton-SOR method ([14]). As it can be appreciated the conclusions were similar to those described along this section.

References

[1] Arnal, J., Migallón, V., Penadés, J.: Synchronous and asynchronous parallel algorithms with overlap for almost linear systems. Lectures Notes in Computer Science **1573** (1999) 142–155.

[2] Bai, Z.: Parallel nonlinear AOR method and its convergence. Computers and Mathematics with Applications **31**(2) (1996) 21–31

[3] Berman, A., Plemmons, R.J.: Nonnegative Matrices in the Mathematical Sciences. Academic Press, New York, third edition (1979). Reprinted by SIAM, Philadelphia (1994).

[4] Bru, R., Elsner, L., Neumann, M.: Models of parallel chaotic iteration methods. Linear Algebra and its Applications **103** (1988) 175–192.

[5] Frommer, A.: Parallel nonlinear multisplitting methods. Numerische Mathematik **56** (1989) 269–282.

[6] Fuster, R., Migallón, V., Penadés, J.: Non-stationary parallel multisplitting AOR methods. Electronic Transactions on Numerical Analysis **4** (1996) 1–13.

[7] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.: PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Tennessee (1994).

[8] IBM Corporation: IBM PVMe for AIX User's Guide and Subroutine Reference. Technical Report GC23-3884-00, IBM Corp., Poughkeepsie, NY, (1995).

[9] Mas, J., Migallón, V., Penadés, J., Szyld, D.B.: Non-stationary parallel relaxed multisplitting methods. Linear Algebra and its Applications **241/243** (1996) 733–748.

[10] O'Leary, D.P., White, R.E.: Multi-splittings of matrices and parallel solution of linear systems. SIAM Journal on Algebraic Discrete Methods **6** (1985) 630–640.

[11] Ortega, J.M., Rheinboldt, W.C.: Iterative Solution of Nonlinear Equations in Several Variables. Academic Press, San Diego (1970).

[12] Sherman, A.: On Newton-iterative methods for the solution of systems of nonlinear equations. SIAM Journal on Numerical Analysis **15** (1978) 755–771.

[13] Varga, R.S.: Matrix Iterative Analysis. Prentice Hall (1962).

[14] White, R.E.: Parallel algorithms for nonlinear problems. SIAM Journal on Algebraic Discrete Methods **7** (1986) 137–149.

Modified Cholesky Factorization of Sparse Matrices on Distributed Memory Systems: Fan-in and Fan-out Algorithms with Reduced Idle Times

María J. Martín[1] and Francisco F. Rivera[2]

[1] Dept. Electrónica y Sistemas, Univ. A Coruña
15071 A Coruña, Spain.
e-mail: mariam@udc.es
[2] Dept. Electrónica y Computación, Univ. Santiago de Compostela
15706 Santiago de Compostela. SPAIN
e-mail:fran@dec.usc.es

Abstract. In this work the fan-in and fan-out algorithms for Cholesky factorization of sparse matrices on distributed memory systems are adapted for modified Cholesky factorization and improved to reduce idle times. The behavior of the new algorithms has been evaluated on two machines with significantly different ratios between processor speed and communications speed, the Fujitsu AP1000 and the Cray T3E.

Keywords: modified Cholesky factorization, sparse matrices, distributed systems

1 Introduction

The modified Cholesky factorization of a symmetric matrix $A \in \mathbf{R}^{n \times n}$ (not necessarily positive definite) is a Cholesky factorization of $A' = A + E = LDL^T$, where E is a non-negative diagonal matrix such that A' is positive definite [6]. This technique is appropriate when the modification of a linear system is justified, as in Newton methods used in nonlinear optimization problems.

The standard Cholesky factorization may be computed in parallel using several methods depending on the access and updating order of the matrices. The main proposals may be classified into three basic types: the fan-out [5], fan-in [1] and multifrontal [7] methods. We present the fan-in and fan-out algorithms for the modified Cholesky factorization on distributed memory systems, together with modified versions reducing processor idle time. Fan-in and fan-out versions for the modified Cholesky factorization on NUMA shared memory systems can be found in [10] and [11].

Recently great efforts have been made to find efficient block oriented implementations of sparse codes on distributed memory systems, examples being the block fan-out method proposed by Rothberg [13], and the block fan-in

method proposed by Dumitrescu et al. [3]. Compared to the column-oriented approaches, block-oriented distributed-memory sparse Cholesky factorization benefits from a reduction in interprocessors communication volume. Unfortunately, block-oriented approaches suffer from poor balance of the computational load and they do not fit for all the matrices, for example, they can not be efficiently applied to random matrices. In any case, the communication pattern generated by a block algorithm is the same as its column counterpart, and therefore, the strategies here presented can be generalized to the block-oriented approaches.

This paper is organized as follows. In Sect. 2 the sequential modified Cholesky factorization algorithm is introduced. In Sect. 3 and 4 the well-known fan-out and fan-in algorithms for parallel modified Cholesky factorization of sparse matrices are briefly looked at and the modifications that reduce idle times are presented. In Sect. 5 the results of trials carried out with a number of sparse matrices on two distributed memory machines with significantly different ratios between processor speed and communications speed, the Fujitsu AP1000 and the Cray T3E, are presented and discussed; and in Sect. 6 our conclusions are summarized.

2   Modified Cholesky Factorization: Sequential Algorithm

Because of the way in which it imposes positive definiteness by addition of a diagonal matrix E, generalized Cholesky factorization is most conveniently treated as a modification of the factorization $A = LDL^T$, where D is a diagonal matrix and L is a lower triangular matrix with ones on the diagonal, rather than as a modification of the standard Cholesky factorization $A = LL^T$ (where there is no constraint on the diagonal of L). E is not calculated separately and added to A before factorization of an explicit matrix $A' = A + E$; instead, it is added implicitly by computing D and L directly from A in such a way that LDL^T is positive definite and the factors are all bounded. This is achieved by ensuring that the elements of D and L satisfy the conditions

$$d_k > \delta \tag{1}$$

$$\|l_{ik}\sqrt{d_k}\| \leq \beta \qquad i > k \tag{2}$$

where δ is a small positive quantity and β is calculated from the largest absolute values of the diagonal and off-diagonal elements of A in such a way as to minimize an upper bound on $\|E\|_\infty$ while ensuring that $E = 0$ if A is "sufficiently" positive definite [6]. In the usual in-place algorithm, L is built up row by row. Sparse matrices are stored using the CSS format, that is a column-wise storage [12]. A sequential algorithm to which sparse matrix techniques are more easily applied, and which is more closely related to the fan-in and fan-out parallel algorithms for Cholesky factorization of sparse matrices, is the following, in which L is built up column by column:

for j=1 to n do

    calculate $\theta_j = max_{s \in \{j+1,...,n\}}\{|a_{sj}|\}$ (3)

    compute $d_j = max\{\delta, |a_{jj}|, \theta_j^2/\beta^2\}$ (4)

    update the diagonal of A for $s > j$:

      $a_{ss} = a_{ss} - a_{sj}^2/d_j$ $s = j+1, ..., n$ (5)

    update off-diagonal elements of A while computing column j of L:

    for $s = j + 1$ **to** n

      $l_{sj} = a_{sj}/d_j$ (6)

      for $k = s + 1$ **to** n

        $a_{ks} = a_{ks} - l_{sj}a_{kj}$ (7)

      endfor

    endfor

endfor

Implementations of modified Cholesky factorization that are intended for the factorization of sparse matrices stored by columns naturally make use of the fact that column s only needs to be updated by column j if $l_{sj} \neq 0$ and there exists some non-zero a_{kj} $(k > s)$. The elimination tree of the matrix provides precise information about dependences among columns [9]. Such implementations will also precede the code shown above with a stage in which, to reduce the fill-in of L, A is reordered (for example, by means of the widely used minimum degree [4] scheme), and by a symbolic factorization stage that determines the pattern of L for the purposes of memory assignment. However, in this paper we concentrate on the actual numerical calculations, which are the most time-consuming.

3 Fan-out Methods

In fan-out methods, computation is data-driven: as a processor receives the data it needs, it progressively computes the diagonal element (d_j) and L_{*j} (the j-th column of L), and as soon as it has completed this task it sends the diagonal and the column to all the processors that require this column to perform modifications. The necessary operations to update columns s $(s > j)$ depending on j are computed on the receiving processors. The algorithm, in a simplified way, is shown in Fig. 1, where $mycols(P)$ is the set of indices of the columns for which processor P is responsible, $ncol(P)$ is initially the cardinality of $mycols(P)$, $nmod(s)$ is initially the number of columns $j < s$ that really need to be used in computing column s, and $users(j)$ is the set of indices of the columns that need column j for their computation; $nmod(j)$ and $users(j)$ can be calculated, before execution of the numerical factorization algorithm, by using the elimination tree of A.

We propose a modification to this algorithm, the fan-out method with premultiplication, in which the computations on each column are performed on the sending processor. If this is done, the number of interprocessor communications is greater than in FO, but the overlap between calculations and communications is increased, and processor idle time is reduced. We propose the algorithm in Fig. 2, where the vector $< j_s >$ is defined by $< j_s > = \{l_{sj}l_{kj}d_j\}_{k \in \{s+1,...,n\}}$.

```
for all j ∈ mycols(P) such that nmod(j) = 0 do
    compute dⱼ (Eq. 3 and 4)
    compute L∗ⱼ (Eq. 6)
    ncol(P) = ncol(P) − 1
    send dⱼ and L∗ⱼ to the processors responsible for the
        columns with indices in users(j)
endfor
while ncol(P) > 0 do
    wait for reception of a column j
    for s ∈ mycols(P) ∩ users(j)
        update column s (see below, Eq. 8)
        nmod(s) = nmod(s) − 1
        if nmod(s) = 0 then
            compute dₛ (Eq. 3 and 4)
            compute L∗ₛ (Eq. 6)
            ncol(P) = ncol(P) − 1
            send dₛ and L∗ₛ to the processors responsible for
                the columns with indices in users(s)
        endif
    endfor
endwhile
```

Fig. 1. The fan-out method (FO) for processor P

Vectors $< j_s >$ for which $s \notin mycols(P)$ are referred to as non-local products, self-messaging has also been suppressed, and in consequence a certain amount of internal traffic control is necessary: if the updating of column s by column $j \in mycols(P(s))$ completes the updating of column s (so making it (almost) ready to be used to update other columns) before column j has finished updating all columns $s^* \in mycols(P(j)) \cap users(j)$, then column s is added to a queue of columns waiting to be used for updating.

4 Fan-in Methods

The main weakness of the fan-out algorithm is the large interprocessor communications volume it involves. The number of interprocessor communications can be reduced if the contributions to column s by all columns j belonging to a single processor $P(j)$ are summed before being sent from $P(j)$ to $P(s)$; this is the idea of the fan-in strategy. The algorithm, in a simplified way, is shown in Fig. 3, where $suppliers(s)$ is the set of column indices j such that $s \in users(j)$, $u(P,s)$ is the vector accumulating updates to column s involving columns $j \in mycols(P)$, and $pmods(s)$ is the number of processors P providing updates to column s.

In FI, columns are computed in order, with the result that high-index columns are not updated at all until all lower-index columns have been computed. To

```
for all j ∈ mycols(P) such that nmod(j) = 0 do
    compute d_j (Eq. 3 and 4)
    compute L_{*j} (Eq. 6)
    ncol(P) = ncol(P) - 1
    compute and send non-local products
    for s ∈ mycols(P) ∩ users(j) do
        compute < j_s > and update column s
        nmod(s) = nmod(s) - 1
    endfor
endfor
while ncol(P) > 0 do
    wait for reception of an update to some column j ∈ mycols(P)
    update column j
    nmod(j) = nmod(j) - 1
    if nmod(j) = 0 then
        compute d_j
        compute L_{*j}
        ncol(P) = ncol(P) - 1
        compute and send non-local products
        for s ∈ mycols(P) ∩ users(j)
            compute < j_s > and update column s
            nmod(s) = nmod(s) - 1
            if nmod(s) = 0 then
                add column s to the queue
            endif
        endfor
    endif
    while queue not empty do
        get next column from queue (column j, say)
        compute d_j
        compute L_{*j}
        ncol(P) = ncol(P) - 1
        compute and send non-local products
        for s ∈ mycols(P) ∩ users(j)
            compute < j_s > and update column s
            nmod(s) = nmod(s) - 1
            if nmod(s) = 0 then
                add column s to the queue
            endif
        endfor
    endwhile
endwhile
```

Fig. 2. The fan-out method with pre-multiplication (PMFO) for processor P

```
for s = 1 to n do
    if s ∈ mycols(P) or ∃ j ∈ mycols(P) ∩ suppliers(s) then
        u(P, s) = 0
        for j ∈ mycols(P) ∩ suppliers(s) do
            u(P, s) = u(P, s) + l_sj(l_sj, ..., l_nj)^T
        endfor
        if s ∈ mycols(P) then
            (l_ss, ..., l_ns)^T = (a_ss, ..., a_ns)^T - u(P, s)
            while pmods(s) ≠ 0 do
                wait for reception of a vector u(P*, s) from some other processor P*
                (l_ss, ..., l_ns)^T = (l_ss, ..., l_ns)^T - u(P*, s)
                pmods(s) = pmods(s) - 1
            endwhile
            compute d_s
            L_*s = L_*s/d_s
        else
            send u(P, s) to P(s)
        endif
    endif
endfor
```

Fig. 3. The fan-in method (FI) for processor P

remedy this, the data-driven fan-in method is proposed, (Fig. 4). This algorithm combines the low message count of the fan-in method with the data-driven character of the fan-out method. This can be achieved by updating columns at the earliest possible moment. The variable $nlmod(s)$ is initialized as $|suppliers(s) \cap mycols(P)|$. DDFI has the same number and volume of interprocessor communications as FI, but the overlap of communications and computations reduces idle times.

5 Experimental Results

The algorithms described above have been implemented on two distributed memory parallel computers, the Fujitsu AP1000 [8] and the Cray T3E [14], some characteristics of which are listed in Table 1. The AP1000 was programmed using its native message-passing routines, and the T3E using the standard MPI library. Double precision floating point arithmetic was used throughout.

For the purpose of evaluating the algorithms, the salient difference between the two computers concerns the ratio between processor speed and interprocessor communications speed: 1.25 FLOPs/byte for the T3E as against only 0.22 FLOPs/byte for double-precision calculations on the AP1000. This difference in the relative capacities of the processing and communications systems means that

```
while ncol(P) ≠ 0 do
    while queue not empty do
        get next column from queue (column j, say)
        compute d_j
        compute L_{*j}
        ncol(P) = ncol(P) - 1
        for s ∈ users(j) do
            nlmod(s) = nlmod(s) - 1
            if nlmod(s) = 0 then
                u(P, s) = 0
                for l ∈ mycols(P) ∩ suppliers(s) do
                    u(P, s) = u(P, s) + l_{sl}(l_{sl}, ..., l_{nl})^T
                endfor
                if s ∈ mycols(P) then
                    (l_{ss}, ..., l_{ns})^T = (a_{ss}, ..., a_{ns})^T - u(P, s)
                    pmods(s) = pmods(s) - 1
                    if pmods(s) = 0 then
                        add column s to the queue
                    endif
                else
                    send u(P, s) to P(s)
                endif
            endif
        endfor
    endwhile
    wait for reception of a vector u(P*, j) (j ∈ mycols(P) )
        from some other processor P*
    (l_{jj}, ..., l_{nj})^T = (l_{jj}, ..., l_{nj})^T - u(P*, j)
    pmods(j) = pmods(j) - 1
    if pmods(j) = 0 then
        add column j to the queue
    endif
endwhile
```

Fig. 4. The data-driven fan-in method (DDFI) for processor P

the attractiveness of the fan-in algorithm relative to the fan-out algorithm is in principle greater for the T3E than for the AP1000.

The performance of the algorithms was evaluated using five benchmark matrices: three belonging to the Harwell-Boeing collection [2] (BCSSTM07, ERIS1176 and ZENIOS) and two randomly generated matrices (RANDOM and RANDOM1). Their characteristics are listed in Table 2, where n_z is is the number of non-zero entries.

Table 1. AP1000 and T3E characteristics

	AP1000	**T3E**
Number of processors	4 to 1024	16 to 2048
Interprocessor networks	Broadcast(50MB/s) 2D torus (25MB/s) Synchronization	3D torus (480 MB/s)
Processor	SPARC	DEC Alpha 21164
Cache memory	128 KB	1st level: 8KB inst./data 2nd level: 96 KB
Local memory	16 MB	64 MB to 2 GB
MFLOPs	8.3 (single precision) 5.6 (double precision)	600

Table 2. Benchmark matrices

MATRIX	n	n_z in A	n_z in L	FLOPs
BCSSTM07	420	3836	14282	579984
ERIS1176	1176	9864	49639	3151680
ZENIOS	2873	15032	62105	4865300
RANDOM	1250	1153	32784	3698402
RANDOM1	2000	1475	106546	23703166

Figure 5 plots, as functions of the number of processors, the number of interprocessor communications involved in the factorization of the matrices by each algorithm.

Figures 6 and 7 show, for the AP1000 and T3E respectively, the total idle time consumed by the last processor in terminating its task during the factorization of the matrices. The idle time for PMFO remains constant or decreases for N greater than about 4; when N is large enough the idle time always seems to be smaller than for FO, as was expected. Similarly, DDFI generally has a slightly smaller idle time than FI (the major exception concerns the factorization of ZENIOS on the AP1000).

Figures 8 and 9 show, for the AP1000 and T3E respectively, the speed-up achieved for the benchmark matrices by each algorithm and for different number of processors. For N greater than a given threshold, PMFO always has better speed-up than FO on the AP1000. On the T3E its extra communications burden outweighs the reduction in processor idle time, at least for $N \leq 16$; in fact, PMFO generally has the best speed-up of all the algorithms on the AP1000 and the worst of all on the T3E. With regard to the fan-in algorithms, the DDFI algorithm improves slightly on FI on both computers.
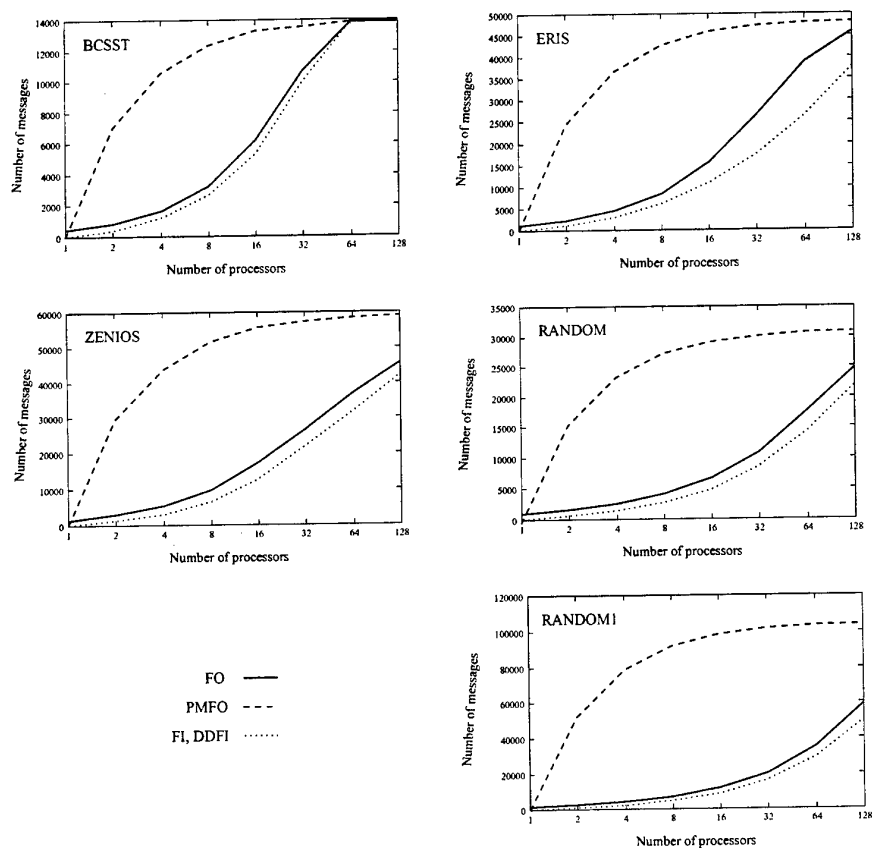
Fig. 5. Number of messages required by the various algorithms
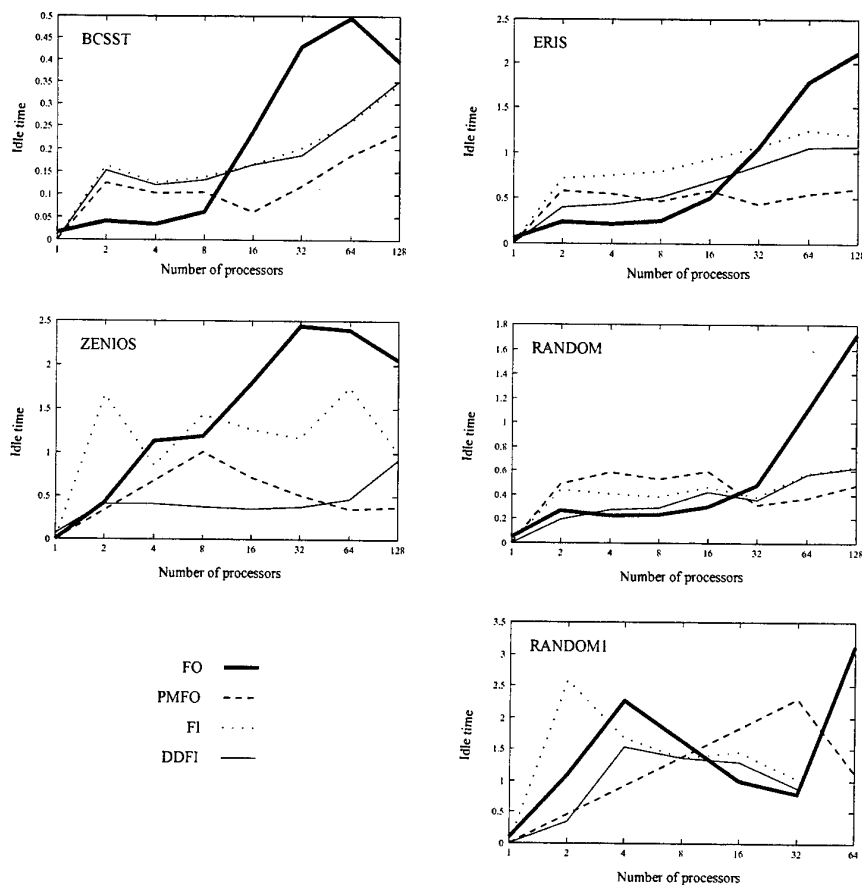
Fig. 6. Idle times, in seconds, on the Fujitsu AP1000
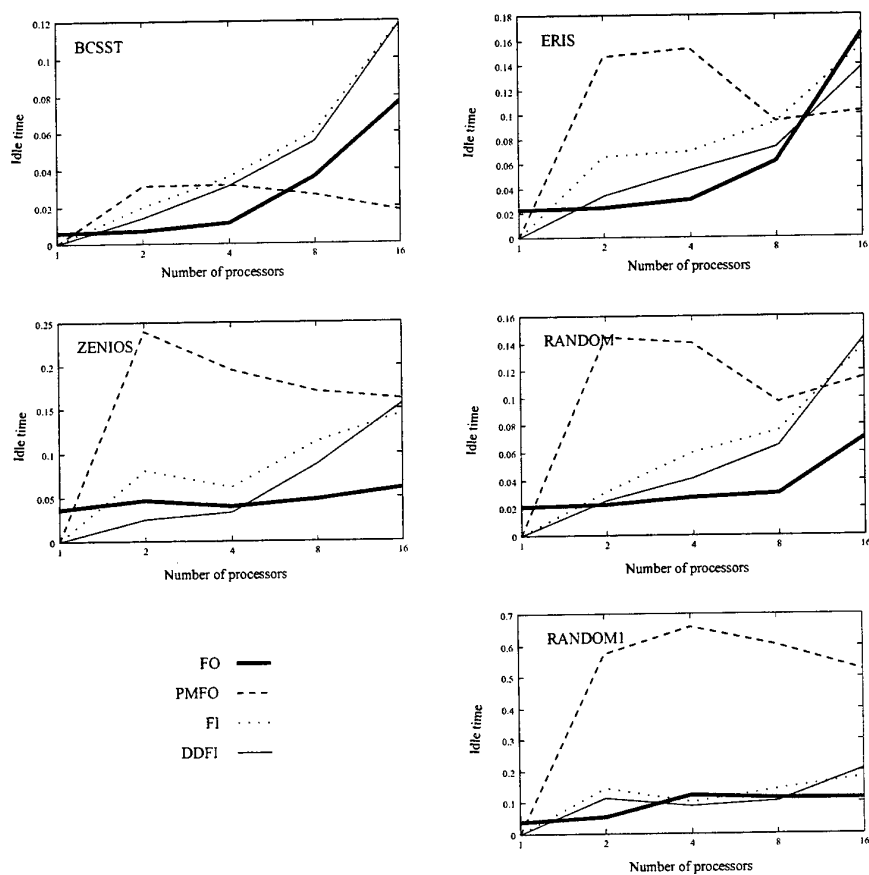
Fig. 7. Idle times, in seconds, on the Cray T3E
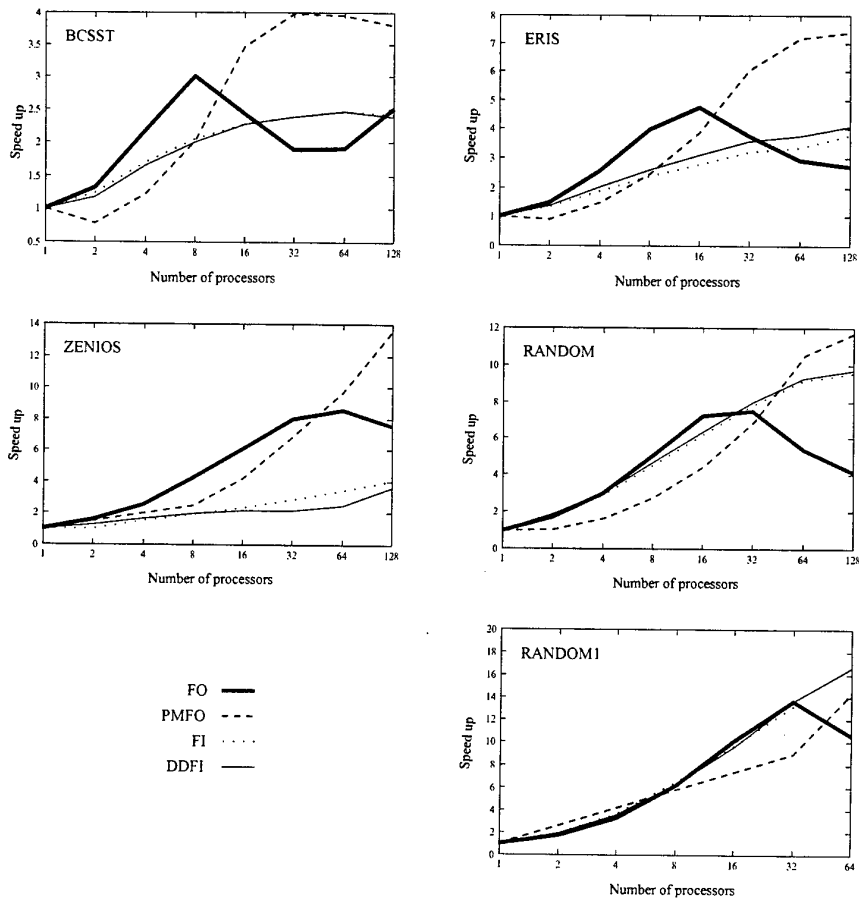
Fig. 8. Speed up on the Fujitsu AP1000
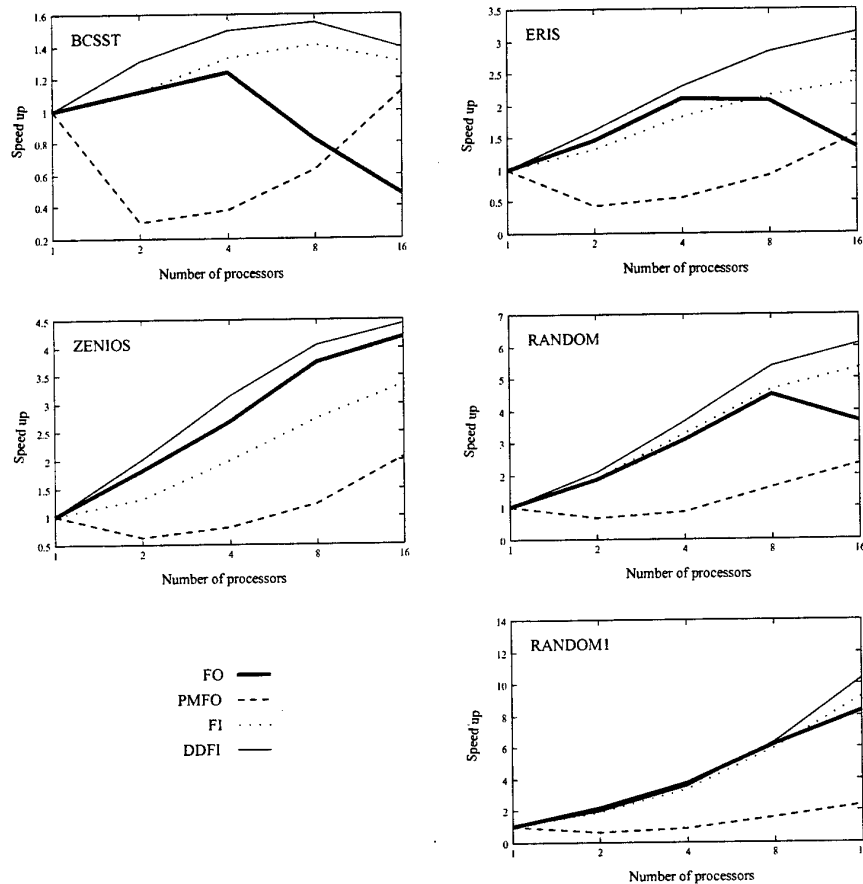
Fig. 9. Speed up on the Cray T3E

6   Conclusions

In this work the fan-in and fan-out algorithms have been adapted for modified Cholesky factorization and improved to reduce idle times. The modified versions generally obtain better results than the unmodified algorithms, except for the case of the modified fan-out algorithm on the T3E. Due to its extra communications burden, this algorithm performs worse than the unmodified algorithm in communications-intensive situations.

The behavior of these algorithms depends on the features of the systems used to execute the codes. The main benefit of the PMFO algorithm is the reduction of idle times at the expenses of an increase in the number of communications. That is why the modification proposed for the FO algorithm (PMFO) is appropriate in

systems where the difference between computation and communication speed is not very high. In those systems in which the communications are a critical factor it would be more convenient to use the algorithms that generate a lower number of communications, that is, the fan-in algorithms. Within the fan-in algorithms, the DDFI algorithm is the one that offers the best performance.

Bearing in mind that the number of floating point operations involved in these calculations is relatively small, the speed-up achieved by these algorithms is quite considerable, even on the T3E.

References

1. C. Ashcraft, S.C. Eisenstat and J.W.H. Liu, A fan-in algorithm for distributed sparse numerical factorization, *SIAM J. Sci. Stat. Comput.* **11** (1990) 593-599.
2. I.S.Duff, R.G.Grimes and J.G.Lewis, User's guide for the harwell-boeing sparse matrix collection, Technical Report TR-PA-92-96, CERFACS, 1992.
3. B. Dumitrescu, M. Doreille, J.-L. Roch and D. Trystram, Two-dimensional block partitionings for the parallel sparse Cholesky factorization, *Numerical Algorithms* **16(1)** (1997) 17-38.
4. J.A. George and J.W.H. Liu, The evolution of the minimum degree ordering algorithm, Technical Report ORNL/TM10452, Oak Ridge National Laboratory, Oak Ridge, Tenn., 1987.
5. A. George, M. Heath, J.W.H. Liu and E. Ng, Sparse Cholesky factorization on a local-memory multiprocessor, *SIAM J. Sci. Statist. Comput.* **9** (1988) 327-340.
6. P.E. Gill, W. Murray and M.H. Wright, *Practical optimization* (Academic Press, London, 1981).
7. A. Gupta and V. Kumar, A scalable parallel algorithm for sparse Cholesky factorization, in: *Proc. Supercomputing'94* , (IEEE Computer Society Press, Washington DC, 1994) 793-802.
8. H. Ishihata, T. Horie and T. Shimizu, Architecture for the AP1000 highly parallel computer, *FUJITSU Scientific & Technical Journal* **29** (1993) 6-14.
9. J.W.H. Liu, The role of elimination trees in sparse factorization, *SIAM Journal on Matrix Anal. Appl.* **11** (1990) 134-172.
10. M.J. Martín, I. Pardines and F.F. Rivera, ıScheduling for algorithms based on elimination trees on NUMA systems, Euro-Par'99, Toulouse (France), pp. 1068-1072, September 1999.
11. M.J. Martín I. Pardines and F.F. Rivera, *Left-looking strategy for the modified Cholesky factorization on NUMA multiprocessors*, Parallel Computing 99 (ParCo99), Delft (The Netherlands), August 1999.
12. S. Pissanetzky, *Sparse matrix technology* (Academic Press, 1984).
13. Edward Rothberg and Anoop Gupta, An efficient block-oriented approach to parallel sparse Cholesky factorization, *SIAM Journal on Scientific Computing,* **15(6)** (1994) 1413-1439.
14. S.L. Scott, Synchronization and communication in the T3E multiprocessor, *ACM SIGPLAN Notices* **31** (1996) 26-36.

An Index Domain for Adaptive Multi-grid Methods[*]

Andreas Schramm

RWCP Parallel and Distributed Systems GMD Laboratory
Kekuléstr. 7, 12489 Berlin, Germany
`schramm@first.gmd.de`

Abstract. It has been known for some time that *groups* as index domains of indexable container types provide a unified view for "geometric" (grids) and "hierarchic" (trees) spatial structures. This conceptual unification is the starting point of further generalizations.

In this paper we present a new kind of index domains that combine both kinds of structure in a single index domain. Together with the "structured-universe approach", these new index domains constitute a framework for an expressive description of adaptive multi-grid discretizations and algorithms.

Keywords: Programming models, data parallelism, container types, structured-universe approach, multi-grid, indexable types, groups.

1 Introduction: Infinite Index Domains and the "Structured-Universe Approach"

As well known, *virtual memory* allows for a dynamic extensibility of data structures like stacks and heaps under preservation of their logical contiguity in the address space. The memory-management unit (MMU) inserts an abstraction layer which maps a finite number of finite substructures (the "pages") of a conceptually infinite address domain ($\mathbb{N}_0$) onto some physical representation.

The *structured-universe approach* is a high-level container type concept with a similar kind of abstraction as virtual memory [12]. Its data types, called "power types", are indexable types with infinite index domains and a distinguished default "zero value" for the element type (0.0 for REAL, etc.).

By appropriate operands and data parallel operations, arbitrary elements of power-type variables can be overwritten, finitely many at a time. Thus, power-type variables always have finitely many non-zero elements (the black "•" in Fig. 1); this property is somewhat reminiscent of infinite-dimensional vector spaces. The state-changing operations can alter finite substructures indexed by chunks of any shape and size and at any location in the index domain (in contrast to the allocation of fixed pages). This allows for a convenient modeling of dynamic and irregular data structures under preservation of their logical contiguity

element type A with zero value "0"

... structured infinite index domain ...

black •'s: arbitrary (finite) support
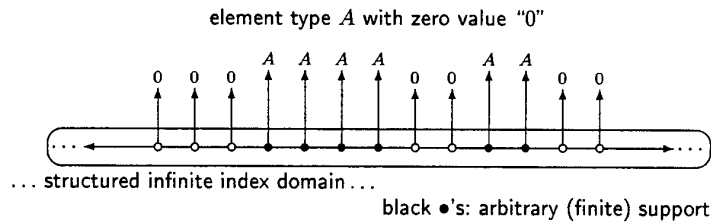
Fig. 1. The "structured-universe approach": Indexable types with infinite index domains and a default "zero" value for the element type. For variables, the supports are restricted to be finite

and neighbourhood structure in their global problem-specific index domain, and leads to compact programs that are close to the problem's underlying mathematical formulae.

Both burden and freedom of setting up the internal technical representation for this "shape-and-granularity polymorphism" are then transferred to the underlying *abstract machine*, which has to act as something like an "Index Domain Management Unit" (in analogy to the MMU). The structural information necessary to do so efficiently on a distributed-memory machine (especially locality information) is contained—partially statically and partially dynamically, depending on the nature of the application—in the index domains, the data and communication patterns, and the operations with them.

An approach of preserving problem-specific structure of index domains is worth as much as the latter indeed have something in them that is worth to be preserved. Therefore the structured-universe approach is equipped with a variety of problem-specific index domains, which are infinite and more general than usual also in other ways to be seen later. A non-obvious example of these index domains is the topic of this paper.

Overview: In Sect. 2 and 3, we analyze the formal properties of index domains in general and for multi-grid data in particular. In Sect. 4 through 6, we sketch a small sample problem, an algorithm, and program text, and summarize the relations between the respective abstract properties of the application and the programming model employed. In Sect. 7 and 8, we make comparisons, summarize, and draw conclusions.

2 What Accounts for the "Right" Index Domain, and Why?

The index domains effect a problem-specific *geometrization* of container data. As for the "right" index domains, for instance we intuitively feel that a two-dimensional grid should be modeled by a two-dimensional array, and that its mapping onto a one-dimensional address space should be done by the compiler. Analogous considerations hold for higher dimensions and, as we shall see, can also

be applied to structures that are usually not perceived as indexable ones, such as trees. A formalization of this intuition leads to following criteria of "naturalness" of index domains:

1. *Nearest-neighbour relations* must correspond to index-arithmetically small distances. Simultaneous nearest-neighbour communications must be "parallel shifts" of data within an index domain.
2. Multiple *non-elemental substructures* of a power-type entity must be indexable meaningfully by multiple *congruent subsets* of the index domain (e.g., the rows in a matrix).
   (Multiple non-elemental substructures occur for instance in routine liftings that express nested parallelism. Multiple substructures of *congruent shapes* correspond to what other container-type concepts express by multiple substructures of the *same type* [10].)

If the structured-universe approach is used with the right index domains, irregularity and dynamicity of spatial structures typically go into the supports of the data (the black "•" in Fig. 1), while the communication patterns and data decomposition schemes retain their regularity in the infinite index domains. Pointers and indirect indexing—which are the structureless "spaghetti" implementation techniques in this field—need to be employed less frequently.

Groups as index domains. It has been known for some time that *finitely generated groups* constitute a unified index domain concept for grids and trees in the sense explained above [5, 10]. The following correspondences hold between spatial structures and the (infinite) groups into which they are embedded as substructures:

$$
\left.
\begin{array}{l}
\text{grids} \subset \text{free Abelian groups} \\
\vdots \\
\text{trees} \subset \text{free groups}
\end{array}
\right|
\begin{array}{l}
+ \\
\\
\textit{degree of commutativity} \\
\\
-
\end{array}
\tag{1}
$$

Now with groups as index domains, the parlance changes a bit:

1. The role of describing "small distances", formerly played by (tuples of) small integers, is now played by (sums of few of) the *generators* of the group.
2. "Parallel shifts" within an index domain, and congruence of subsets, are defined in terms of the respective *group operation*, here generically written "$\oplus$".

For integer grids, these terms still coincide with the intuitive understanding. Analogously for trees, the neighbourhoods characterized by the generators of the group are those between parent and child, and communications between parents and their respective childs are described by parallel shifts of data within the index domain by a small index-arithmetic distance. The non-commutativity of the corresponding groups reflects the special geometry of trees, which after all is different from that of grids.

In short, groups as index domains constitute a unification of container structures that are commonly regarded as quite different. And even better, this

$$d = down, \ e = east$$
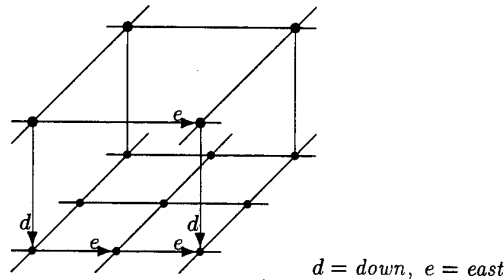
Fig. 2. Section of an example group that models the geometry of multi-grid discretizations. Observe the close interaction of the grid-like and the tree-like spatial structures, as formalized by relation (2)

unification is the starting point for a further generalization, which we now begin to introduce.

Degree of commutativity. We begin with a remark about commutativity of groups. There are several ways to attribute a gradated "degree of commutativity" to groups, as opposed to a mere Abelian-or-not classification. In all of these ways, Abelian groups and free groups mark the opposite extreme cases. So it appears to be natural to investigate whether "intermediate" groups between the extremes serve some purpose. This is indeed the case, and one of the possibilities to fill in the ellipsis in (1) is the kind of groups we are going to present in the next section. It is not much of a surprise that this kind of groups exhibits an amalgamation of both grid-like and tree-like spatial structures in the same index domain.

3   The Index Domain for Multi-grid Data

Multi-level methods (methods that employ multi-level discretizations) occur in various fields. They are renowned for their efficiency and, in the case of the dynamically adaptive variant on distributed-memory machines, notorious for their difficulty of programming. They are treated in more depth e.g. in [2,6]; here we just mention that their characteristic property is the combined use of discretizations of the same physical space at different levels of resolution. The algorithms typically employ both intra-level and inter-level communications. Here we confine ourselves to geometric multi-grid methods.

Our starting point is the observation that the spatial resolution of the discretization (usually) *doubles* in the transition from one level to the next one. For an illustration we assume a two-dimensional integer grid (index domain $\mathbb{Z}^2$) and use the term "one level down" for the transition to the next level with doubled resolution. Then, in order to cover a certain distance x at one level farther down, we have to go *twice* as many steps. (E.g., first going east one step and then going down is the same as going down first and then going east *two* steps; see Fig. 2).

This observation can very well be formalized as a *relation within a non-Abelian group:*

$$x \oplus down = down \oplus x \oplus x \qquad \text{for all } x \in \mathbb{Z}^2 \ . \qquad (2)$$

So we construct the index domain for a multi-level discretization of a two-dimensional domain as follows: The group $\mathbb{Z}^2$ is extended by an additional generator, called "down", and made subject to the relation (2). Figure 2 shows a section of the resulting index domain, which clearly exhibits the desired multi-level nature.

We compare the above-mentioned communication relations in multi-level methods with the geometry represented by this group[1]: *Intra-level* nearest-neighbour communications (e.g., in the computation of point-wise residuals) work just as in integer grids. *Inter-level* communications (e.g., in the computation of prolongation and restriction operators) can be expressed *in the same way* by data shifts by small distances, using *down* or its inverse, respectively.

In summary, the presented index domain is capable of formalizing *both* kinds of locality of originally different nature. Hence, both kinds of (translation-invariant) communication can be expressed as convolutions by appropriate stencils. The only difference is that the convolution takes place in the new kind of index domain and is defined by means of the group operation "$\oplus$".

Groups that model the geometry of *anisotropic* (nonstandard) coarsenings can be constructed similarly, but this is not carried out here.

4   A Sample Problem and its Numerical Method

4.1   The Problem

The motivations for multi-level approaches are (i) faster convergence, and (ii) adaptive refinements, for a reconciliation of computational effort and accuracy.

As example for both the structured-universe approach and the new kind of index domains, we present an adaptive multi-grid application. We consider a simple boundary-value problem. We assume as given

$$\text{a domain} \quad \Omega = (a,b) \times (a,b) \subset \mathbb{R}^2$$
$$\text{a function} \quad f : \Omega \to \mathbb{R}$$
$$\text{a boundary function} \quad F : \delta\Omega \to \mathbb{R}$$

and seek as solution

$$u : \overline{\Omega} \to \mathbb{R}$$
$$\text{with} \quad Lu = -\Delta u = f \qquad \text{on } \Omega \qquad (3)$$
$$\text{and} \quad u|_{\delta\Omega} = F \ .$$

---

[1] Recall that the purpose of the index domains in the structured-universe approach is to express the "natural" problem-specific neighbourhoods and congruences within container data, as explained in Section 2.

We assume that the right-hand side f possesses a singularity somewhere on the boundary $\delta\Omega$, so that the problem calls for adaptive refinement.

4.2   The Numerical Method

Data fields and basic operations. For an initial coarse level 0 and for a finite number of successively finer levels, the following infinite-grid quantities with finite supports are maintained: "interpolated solution", "solution corrector", "residual", and "right-hand-side perturbation"; these names may appear abbreviated in equations and program text. Figure 3 sketches the data structure and the data flows therein.

The residual follows the other quantities so that the following variant of (3) is fulfilled (in its respective discretized form):

$$L(interpol.\_solution + soln.\_corrector) = f + RHS\_perturbation + residual \quad (4)$$

The solution algorithm will be constructed from the following four basic operations (larger level numbers correspond to finer resolutions):

1. *Initialization* at level 0: At the coarsest level, the (small) system of equations is solved, and the solution is stored into the field *interpolated_solution.*
2. *Interpolation* from level k to $k+1$: A suitable interpolation operator is applied to the sum *interpolated_solution + solution_corrector* of level k, and the result is stored into the field *interpolated_solution* of level $k + 1$.
3. *Smoothing* at a level k: A smoothing method is applied to the residual at level k, and the resulting correction values are added to the already existing *solution_corrector.* (The residual decreases accordingly.)
4. *Restriction* (residual coarsening) from level $k+1$ to k: A restriction operator is applied to the residual at level $k+1$, and the result is stored into the field *RHS_perturbation* of level k.

Organization of the basic operations. In the multi-grid terminology, the method presented here is a *full multi-grid* (FMG) scheme with V(1,1)-cycles. It is organized as follows: After the initialization at level 0, the process *descends* (i.e., interpolation followed by smoothing) to a certain maximal depth and then *ascends* (i.e., restriction followed by smoothing) back to level 0. These descents and ascents are continued with a successively increasing maximal depth until no further refinement is necessary.

For simplicity of presentation, the algorithm presented here deviates somewhat from the conventional ones by the following modifications: We neglect the fact that usually different interpolation operators are employed in the FMG refinements and the multi-grid cycles. Second, the coarse-grid corrections are calculated for the perturbed original equation (this is the *full approximation scheme* used for non-linear equations), and not from the pure defect equation. Third, the coarse-grid corrections of refined subgrids nevertheless take place in
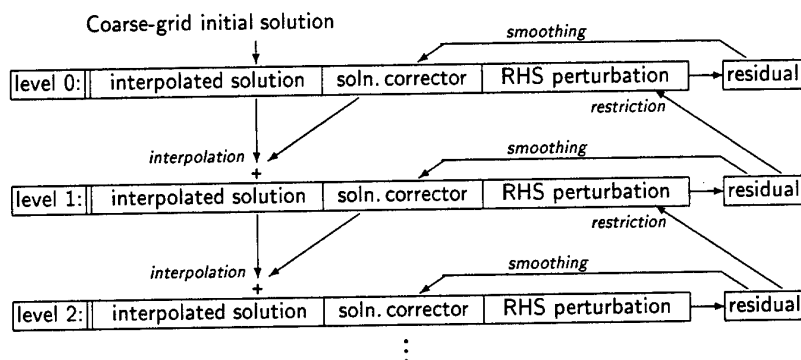
Fig. 3. Data fields and dependences for the modified full multi-grid (FMG) scheme. Larger level numbers refer to finer grids. With spatial adaptivity, some finer levels may represent only subsets of the problem domain

the larger subregions pertaining to the coarser grids. This appears to be more intuitive, as even a residual with a limited support may very well lead to a *global* correction of the solution.

Spatial adaptivity consists in the technique that increasingly finer resolutions (with larger computational effort) are applied only to increasingly smaller *subregions* of the problem domain, under control of some refinement criterion (a local discretization-error estimator). These subregions turn out to be the neighbourhoods of the singularity of the right-hand side f of (3). (We assume that f possesses only one singularity, so that the latter can be enclosed at each level by a single rectangular subdomain.)

FMG, if used with a sufficiently good interpolation operator, has the property that for each level of refinement, an accuracy up to the corresponding discretization error of that level is achieved already after a single multi-grid cycle. We exploit this property and consider convergence to have occurred too when no further local refinement is required.

5 The Program

5.1 Prerequisites and Basic Program Patterns

The program will be presented in an experimental linguistic concretization of UNIVERSE [11] on the top of Oberon-2 [9]. Keywords and the predefined identifiers of the host language are in all-caps. For space considerations, only very brief explanations are given here.

An operation pattern that will occur frequently in the program text is the following one

$$power\_type\_variable[subdomain] := power\_type\_value \ \$\$ \ power\_type\_value; \quad (5)$$

typically such that one operand of the infix operator "$\$\$$" (convolution) identifies a static communication pattern (a stencil).

By such statements, some elements of a power-type variable are overwritten, viz. those elements that are indexed by the subdomain on index position, the "selection mask". In a writing access like here, masking of a power-type *variable* means that only the selected elements are overwritten; masking of a power-type *value* means that the non-selected elements are replaced by zero in the result.

The subdomain expressions in the program text to follow might appear quite complicated at first sight, but they resemble the conventional mathematical notations of intervals, element-wise sums of sets, Cartesian products, etc.

The infix expression on the right-hand side of the assignment is a (discrete) *convolution* product (a shortcut for "$\$ * \text{REDUCE BY} + \$$" if the element types of the operands are numbers). It yields a result with the same index domain as the operands, and for all non-zero elements x_i of the first operand and y_j of the second operand, the products $x_i * y_j$ are accumulated into the element $z_{i \oplus j}$ of the result z.

Convolutions are the method of choice for the (non-redundant) expression of *translation-invariant* communications (data movements) *within* an index domain. In all usages here, one of the operands is a static pattern (a stencil), which represents the discretization of the underlying linear operator.

A sensible implementation will compute only those elements of the right-hand side that are actually used (e.g., not masked out). In order to facilitate this, power-type products (e.g., the convolution) and also implicit liftings of scalar pure functions and operators have lazy semantics in UNIVERSE.

5.2 Global Declarations

First, two index domain are declared, viz. the two-dimensional infinite integer grid and the index domain of Sect. 3.

The INDEXCOUNTER declaration declares two symbolic power-type constants Xcoord and Ycoord with index domain $\mathbb{Z} \times \mathbb{Z}$ and element type $\mathbb{Z}$. These constants provide the "canonic" x and y coordinates of the integer grid; after a multiplication by the appropriate mesh size they will be used to parametrize the parallel invocations of the right-hand side f and of the boundary condition F. For every index point that can be written as sum of $(1,0)$ and $(0,1)$ and their inverses, the respective associated symbolic counter indicates how many of the generators are used to express that index point.

The variable values holds the data structure depicted in Fig. 3; for each level, regions[level] holds the integral corner coordinates of the finite rectangular subgrids that correspond to Ω or its refining subregions, respectively.

```
INDEXDOMAIN
   PlaneGrid = Z x Z;
   MultiGrid = EXT(PlaneGrid, Down, 2);   (* see Sect. 3 *)
INDEXCOUNTER Xcoord OF (1,0), Ycoord OF (0,1);
TYPE
   Point: RECORD sol, corr, resid, perturb: REAL END;
```

```
    RectRegion: RECORD xa, xz, ya, yz, num: INTEGER END;
VAR
    values: [MultiGrid]<Point>;
    regions: [ZZ]<RectRegion>;
```

5.3 The Basic Operations

The solution of the small system of equations at the coarsest level (Step 1 in Sect. 4.2) is often done with a direct solver, and is not shown here.

Residual evaluation. The residual is evaluated according to (4). Besides of point-wise real additions and subtractions, the computation consists of the evaluation of the discretized form of L and of the right-hand side f. The former is done by convolution by the stencil `Lstencil`, which is a power-type constant with index domain $ZZ \times ZZ$, given below as a cascaded conditional expression. The (scalar) function f is invoked multiple times ("lifted") with an explicitly specified replication space appearing before it, and each invocation accesses the corresponding elements of the power-type arguments, which are obtained from the symbolic integer coordinates `Xcoord` and `Ycoord` scaled by the mesh size h.

```
CONST  Lstencil =
    {(0,0)} => 4.0 :
    {(0,1),(1,0),(0,-1),(-1,0)} => -1.0;
```
$$\left(* \begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix} *\right)$$

```
PROCEDURE  compResidual(level, xa, xz, ya, yz: INTEGER);
    VAR h: REAL;
    BEGIN
      h := 1.0 / (2**level);
      (*  evaluation of residual according to (4): *)
      values[{level*down}⊕{xa+1..xz-1}×{ya+1..yz-1}].resid :=
        (values.sol+values.corr) $$ Lstencil/(h*h)
        - values.perturb
        - {level*down} $$ [{xa+1..xz-1}×{ya+1..yz-1}].f(Xcoord*h,Ycoord*h)
    END compResidual;
```

Smoothing. Smoothing is done by *red-black relaxation*, which combines good smoothing properties with good parallelism properties [16]. The term refers to the colouring of a grid in a chequerboard pattern: First, all "red" points are relaxed, which can be done in parallel, and then all "black" points, again in parallel (observe the two subdomains `RedGrid` and `BlackGrid` in the following program fragment). As usual, "relaxing a grid point" refers to the point-wise error smoothing by averaging that grid point with its neighbours, as determined by the stencil involved, with taking into account the right-hand side at the same coordinates.

```
SUBDOMAIN
    RedGrid = { * OF (2,0),(1,1)};   (* spans all even-parity points *)
    BlackGrid = {(0,1)} ⊕ RedGrid;   (* coset of RedGrid *)

PROCEDURE smooth(level: INTEGER);
    VAR xa, xz, ya, yz: INTEGER; h: REAL;
    BEGIN
        xa := regions[level].xa; xz := regions[level].xz;
        ya := regions[level].ya; yz := regions[level].yz;
        h := 1.0 / (2**level);
        compResidual(level, xa, xz, ya, yz);
        values[{level*down}⊕RedGrid].corr :=
          values.corr + values.resid*h*h/4.0;
        compResidual(level, xa, xz, ya, yz);
        values[{level*down}⊕BlackGrid].corr :=
          values.corr + values.resid*h*h/4.0
    END smooth;
```

Restriction. A customary and robust method for restriction (coarsening) of residuals is "full weighting" [2, 6]. Every point of the coarser grid gets assigned a weighted sum of several nearby points of the finer grid, and the weights are set up in such a way that all points in the finer grid—also the interleaving ones— have the same total sum of weights, i.e., the same "influence" on the coarser grid.

```
CONST Restrictor =
    {-down}   => 4.0/16.0 :
    {(0,1),(1,0),(0,-1), (-1,0)}⊕{-down} => 2.0/16.0 :
    {(1,1),(-1,1),(1,-1),(-1,-1)}⊕{-down} => 1.0/16.0;

PROCEDURE restrictResid(level:INTEGER);
BEGIN
    values[{level*down}⊕PlaneGrid].perturb :=
      values.resid $$ Restrictor;
END restrictResid;
```

The remaining steps. The remaining steps are explained only in passing.

The interpolation is in principle a linear operator just like the restriction, expressed by convolution by a stencil. However, two details have to be taken into account: (i) on the boundary $\delta\Omega$, the solution candidate should be computed directly from the given boundary function F, and not by interpolation. (ii) Cubic interpolation—which is advisable in FMG for numerical reasons—requires "four points in a row", but near boundaries and corners, these four points are not available in a symmetric distribution, i.e., two at either side. Therefore, different interpolation patterns have to be used near boundaries and corners. Both of these detail case discriminations can be expressed combining several assignments like (5) with different subdomains.

The procedure `SetupRectangle(level)` determines the corner coordinates of the domain rectangle of that level and stores them into the fields of the variable `regions[level]`. This is done either in accordance to $\Omega = (a, b)^2$ at those levels where the entire domain Ω is to be considered, or according to a local error estimator at those levels where adaptive refinement is to be employed. The field `regions[level].num` is set to 0 iff the refinement area is empty.

5.4 The Main Program

The main program implements the algorithm sketched in Subsect. 4.2. The algorithm begins at the coarsest level 0 and terminates at some fine level, viz. when the refinement criterion states that no further refinement is necessary.

```
VAR level, depth, maxdepth, i: INTEGER;
  ...
SetupRectangle(0);
(* initial solution at level 0 (basic operation #1). *)
(* FMG multi-grid iteration: *)
maxdepth := 1;
LOOP
  (* descend down to level maxdepth: *)
  level := 0;
  REPEAT INC(level);
    interpolate(level); smooth(level)
  UNTIL level = maxdepth;
  interpolate(level+1);
  SetupRectangle(level+1);
  IF regions[level+1].num = 0 THEN EXIT (* from LOOP *) END;
  INC(maxdepth);   (* for the next round *)
  (* ascend back to level 0: *)
  REPEAT DEC(level);
    restrictResid(level); smooth(level)
  UNTIL level = 0;
  FOR i:= 1 TO ... DO smooth(0) END; (*few more smoothings at lev. 0*)
END (* LOOP *);
```

6 Observations

We summarize and generalize the key observations about the relations between numeric applications and high-level programming models:

– Spatial discretizations with arbitrary refinements are modeled naturally by countably infinite-dimensional vector spaces. Problem-specific operators (e.g., differential, prolongation, and interpolation operators) often are linear operators on these vector spaces.
A programming model that models such applications in terms of vector spaces and linear operators can be expected to lead to compact programs.

- The phenomenon of irregularity and dynamicity of spatial structures is banned from the semantics—there are no "irregular" vector spaces—and delegated to the system.
- If the canonic bases for these vector spaces are chosen adequately, then the problem-specific linear operators correspond to "simple" (e.g., nearest-neighbour and/or translation-invariant) communication patterns.
  A programming model that provides index domains that reflect the locality properties of the application can be expected to lead to efficiency on distributed-memory parallel machines.
- In the case of geometric multi-grid discretizations, the interaction of grid-like and tree-like geometries in the same index domain can be modeled by a group with appropriate equality relations.

7   Comparisons

Here we confine ourselves to a few other programming models that are related to the modeling of spatial structure of parallel applications. For a broader survey, see for instance [15].

Other models with indexable types. A now "classic" programming model that elaborates on indexable types is *Crystal* [3]. Crystal is a higher-order functional language with data fields over generalized index domains, such as grids, trees, and hypercubes, and data-field and index-domain morphisms. The semantic complexity of Crystal is considerably higher than that of UNIVERSE.

Groups as index domains have also been proposed for the programming model $8\frac{1}{2}$ [5]. $8\frac{1}{2}$ does identify the correspondence between generators of groups and basic neighbourhood structures (Cayley graphs), but does not further pursue the issue of non-Abelian groups and the identification of useful ones, and proposes their representation by libraries.

More general type systems. There are other parallel programming models that employ inductive types or even more general settings for the modeling of spatial structure. As examples we mention the *Bird-Meertens formalism* [13] and *NESL* [1] for (join-) lists and *Categorical Data Types* [14] for polymorphic trees. A typical property of the category-theoretic approach is the inference of the container decompositions from the type constructors. There also is a category-theoretic understanding of *shapes* [8] (by which UNIVERSE simply understands patterns in structured infinite index domains).

An abstract generic concept of capturing parallelism is that of algorithmic *skeletons* [15]. Programs are composed from as few as possible predefined parametrizable building blocks (typically a small set of second-order functions), aiming at implementing parallelism as composition of pre-implemented internally parallel algorithmic fragments. Formally, also the power-type products and procedure liftings of UNIVERSE constitute such a small set of second-order functions

that systematizes parallel access patterns for indexable container types. But in contrast to "plain" skeleton concepts, UNIVERSE encodes the knowledge about the problem geometry *also*—if not primarily—into index domains and shapes of subdomains and operands—perhaps a sort of "geometry skeletons". The free combination and interaction of these two concepts makes an implementation of UNIVERSE a demanding task and weakens its simplicity as a skeleton concept.

More technical approaches. There are numerous approaches whose philosophy differs from the author's in that they consist in *implementation directives* for some abstract machine, as opposed to expressing structural information about the applications *in the semantics*. To this class belong data partitioning/distribution algebras, languages, and systems, also High-Performance Fortran [7]. Another approach, a template concept for the modeling of irregular spatial structures, is given in [4].

8 Summary and Conclusion

We have mentioned the structured-universe approach, a container-type concept based on structured infinite index domains. We have mentioned the known fact that groups as index domains are general enough to host grids as well as trees, and to formalize their different geometries under a unified scheme. We have exploited this generality of groups as index domains further and have introduced a new kind of groups to host multi-grid algorithms. These groups reflect the multi-level nature in that grid-like and tree-like neighbourhoods interact in the same index domain. We have related this phenomenon to commutativity properties.

This result sheds some more light on the little recognized versatility of (possibly non-Abelian) groups as spatial domains. Originally conceived as a unification of two different kinds of spatial structure, they generalize further to an "interpolation" between these two. Together with the "structured-universe approach" —an abstraction scheme reminiscent of infinite-dimensional vector spaces over geometrically structured index domains—this new kind of index domains provides an expressive formalization framework for adaptive multi-grid algorithms. Such formalizations are a prerequisite for the high-level programming of distributed-memory machines by compact programs, and may constitute the input for an efficient automatic mapping onto such machines.

References

1. G. E. Blelloch. NESL: A nested data-parallel language. Technical report CMU-CS-93-129, Carnegie-Mellon University, Pittsburgh, PA, 1993.

2. A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Math. Comp.*, 31:333–390, 1977.

3. M. Chen, Y. Choo, and J. Li. Crystal: Theory and pragmatics of generating efficient parallel code. In B. K. Szymanski, editor. *Parallel Functional Languages and Compilers*, Chapter 7, pages 255–308. Addison-Wesley Publishing Company, Inc., 1991.

4. M. Gerndt. Unstructured templates for programming irregular grid applications on high performance computers. *Parallel Computing: Fundamentals, Applications and New Directions. Advances in Parallel Computing.* 12:251–260, 1998.

5. J. L. Giavitto, O. Michel, and J. P. Sansonnet. Group-based fields. In T. Ito, R. H. Halstead, Jr., and C. Queinnec, editors, *Parallel Symbolic Languages and Systems, International Workshop PSLS'95, Beaune, France, October 1995, Proceedings*, number 1068 in Lecture Notes in Computer Sciences, pages 209–215. Springer-Verlag, 1996.

6. W. Hackbusch. *Multi-grid Methods and Applications.* Springer, Berlin, Heidelberg, New York, 1985.

7. High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1, 2):1–170, 1993.

8. C. B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, Dec. 1995.

9. H. Mössenböck. The programming language Oberon-2. Technical report #160, ETH, Zürich, 1991.

10. A. Schramm. Formalization of spatial structure. RWC Technical Report TR-98-011, Real World Computing Partnership, Japan, 1999.

11. A. Schramm. *The Programming Language Extension* UNIVERSE *(provisional Oberon version)*, 1999. http://www.first.gmd.de/~schramm/UNIVERSE_Oberon.ps.gz.

12. A. Schramm. The "structured-universe approach" for irregular and dynamic spatial structures. In *Proceedings of HPCS'99, Kingston, Ontario, Canada.* Kluwer Academic Press, 1999. To appear.

13. D. B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–51, Dec. 1990.

14. D. B. Skillicorn. *Foundations of Parallel Programming.* Number 6 in Cambridge International Series on Parallel Computation. Cambridge University Press, Cambridge, 1994.

15. D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

16. U. Trottenberg and K. Oosterlee. Parallel adaptive multigrid — an elementary introduction. Arbeitspapiere der GMD 1026, GMD – Forschungszentrum Informationstechnik GmbH, Oct. 1996. ISSN 0723-0508.

PARADEIS: An STL Extension for Data Parallel Sparse Matrix Computation

Franck Delaplace and Didier Remy

La.M.I., C.N.R.S. EP 738
Université d'Evry Val d'Essonne, Bld F. Mitterand
91025 Evry Cedex, France
{delapla,remy}@lami.univ-evry.fr
phone: (33) 1 69 47 74 63 – Fax: (33) 1 69 47 74 72

Abstract. In this paper we will present the main basis of PARADEIS which extends the features of the STL library to deal with data-parallelism and sparse matrix computation.
topics: Languages and Tools, Numerical methods

1 Introduction

Sparse matrix computation is recognized to be ubiquitous in computational science but the parallel programs are error-prone, hard to debug, and difficult to maintain. Consequently, the simplification of these parallel programs represents an emerging trend.

In this context, several issues can be considered. The first issue puts the emphasis to a compilation based approach. Sparse compiler automatically restructures a dense program dealing with arrays to a sparse program dealing with sparse arrays [3], [5].

The second issue is to consider a run-time support based on a numerical library which offers a set of numerical programs in order to cover the main features of sparse matrix algorithms.

At the cross-roads of this two issues, we propose a run-time support which provides a set of basic operations to deal with sparse matrix structure. The goal is to define an abstract data structure which eases the parallel programming of sparse matrices. It aims at extending the STL library according to this scope. One of the goal of PARADEIS is to be used as a user-library as well as a back-end of parallel sparse compiler. The extensions are mainly focused on two topics : data-parallelism and sparsity management.

The extended abstract is organized as follows : the Section 2 describes the main features of PARADEIS. The Section 3 briefly outlines the description of the parallel sparse structure. The Section 4 shows experiment on efficiency of running PARADEIS programs. Finally, the Section 5 will conclude this paper by a discussion.

2 STL Extension

The Standard Template Library provides a set of well structured generic C++ components that work together in a seamless way. The library con-

tains three main components : *container* which manages set of memory location, *iterator* which provides a means for an algorithm to traverse through a *container* and *algorithms* which are computational procedures. The extensions concern the two first components, namely container and iterators. They cover the parallelism and the sparsity management. In addition, we define a collective communication primitive.

2.1 Sparse Computation Extension

This section will concern the extensions for sparse computation, they aim at hiding the sparsity to users to provide a simple programming framework. The sparsity management is dealt by methods which operate on a dedicated container named *SparseArray*. Throughout this paper, we will examplified the features with the algorithm of vector addition $R = V_1 + V_2$ where the vectors V_1 and V_2 are sparse and R is dense. The SparseArray container provides an homogeneous framework for dense and sparse array without loss of performances for dense array computation. So, the declaration will be

```
SparseArray V1(100), V2(100), R(100);
```

Numerous storage formats have been proposed in sparse-matrix literature, for our work, we have generalized the Block storage format to a Distributed Block storage format. But, this representation is hidden to programmers so the program is generic. Thus, the internal storage format can be changed without any modification of the user-program.

Iterators Iterators are a generalization of pointers that allow a programmer to work with different data structures in a uniform manner. The iterators are divided into several classes. Each class corresponds to the capabilities of an iterator. The class of *forward* iterators scans all the structure. In PARADEIS, the *sparse forward* iterator will scan only the non-zero value. A general scheme of an assignment restricted to entries-structure follows a dense programming style :

```
Iterator i(R);
for(i.begin(); i.end(); i.next())
    R[i] = ...;
```

Another mechanism is added to complete the requirement of sparse computation. This mechanism synchronizes the iterators to a selected structure of entries of a sparse data-structure. For example, if we consider, an addition of two sparse vectors $R = V_1 + V_2$; the ith components of R will be the result of the addition of the ith components of V_1 and V_2. But, in the sparse representation, the ith component of each vector are not located in the same place. Moreover, one of the component may not exist at all. The synchronized iterators will have the same behavior as the dense computation in vectors addition. Assuming that the entries structure of R is properly declared as the union of the two entries structures V_1 and V_2, the iterators scanning V_1 and V_2 are synchronized to the space of the entries of R. A comparison is performed to determine if the current pointed value of V_1 and V_2 represent the same entries. If it is the case then the addition is performed. If one of the entries is missing

the corresponding value is 0. Except for the declaration of the iterators, first the management of these rules is hidden to the programmer and second the program is close to the dense program :

```
Iterator i(R), i1(V1,i), i2(V2,i);
for(i.begin(), i1.begin(), i2.begin(); i.end();
    i.next(), i1.next(), i2.next())
  R[i] = V1[i1] + V2[i2];
```

Entries Set Operations In conjunction with synchronized iterators, a set of primitives is provided to operate on the structure and not on the values. It will offer the capabilities to symbolically determine the shape of the expected structure at the end of the numerical computation.

It has been shown in [1] that a correspondence can be achieved between arithmetic operators and set/logical operators. For the vectors addition, the sparse structure of the entries of R is defined as the union of the entries of V_1 and V_2. Then, this structure is used as a reference pattern for synchronizations of the iterators.

```
SparseArray R(union(V1, V2)); // R is assumed sparse here
```

More complex algorithms can be used to determine the structure of the entries as the symbolic factorization in the sparse Cholesky factorization. But these algorithms can be expressed by set and logical operations. In this context, PARADEIS provides basic functions to symbolically compute the fill-in introduced during the computation.

2.2   Data-parallel Extension

In this section, we describe the extension of iterators for parallel sparse-matrix computation. Before explaining its semantic, we will introduce the context of the execution. Programs are written in a SPMD style. The target architecture is a distributed memory architecture or a cluster of workstations. The sparse matrix is folded to processors. And in each processor, the computation is applied to the local part of the data.

Sparse Parallel Iterators The parallel execution will be expressed by an iterator named DoAllIterator. Its semantic will guarantee that every elements in the sparse matrix will be scanned, but in any order. Its semantics is derived from the standard conditions of parallelization which aims at relaxing constraints on the sequential execution order.

Pragmatically, the iterator scans only the local part of the data on each processor. The conversion from a global to local address is handled by an iterator's method. Under the assumption that vectors are appropriately distributed, the parallel sparse program of vectors addition is :

```
SparseArray V1(100), V2(100), R(100); // R is dense
// Initialization of V1 and V2
DoAllIterator i(R), i1(V1,i), i2(V2,i);
for(i.begin(), i1.begin(), i2.begin(); i.end();
    i.next(), i1.next(), i2.next())
  R[i] = V1[i1] + V2[i2];
```

Communication Primitives The communication primitives are necessary to exchange values between processors. Since PARADEIS is based on a run-time process, no compiler automatically transforms accesses to global memory to communication. So communications are explicit.

In [4], which is focussed on the communication support of the exchange, we provide more details about the communication scheme.

The design of the program follows the rules of the BSP model [6] : The communication phases are separated from the execution phases.

But, in order to simplify the programmer task, the communications primitives correspond to a global communication. So the program is the same for the emitter and the receiver and communications defined according to a global address space. In some extend, it may correspond to an alignment of values.

For instance, given the following assignment in Fortran 90, X(1:100) = Y(2:101), if we assume that the array X and the array Y have the same distribution, a communication must be achieved. In PARADEIS, this communication is defined by the **exchange** primitive as follows :

```
Y.exchange(X, Section(1,100), Section(2,101));
```

The communication primitive manages the sparsity as well as the distribution. For sparse arrays, every values of Y contained in the interval 2 : 101 are exchanged. The exchange is "aligned" to the sparse structure of X. The exchange primitive also broadcasts values. For instance, given Fortran 90 statements X(1:100,1:100) = Y(1:100), the communication will be expressed as follows :

```
Y.exchange(X, Block(Section(1,100), Section(1,100)),
              Block(Section(1,100), EXTEND)));
```

The **EXTEND** keyword specifies that Y must be extended in dimension before the exchange is performed . The extension in dimension is virtual and it does not waste memory space.

The exchange has been performed by an inspector-executor scheme. Then, the vector addition with an inappropriate distribution is :

```
SparseArray V1(100), V2(100), R(100); // R is dense
DoAllIterator i(R), i1(V1,i), i2(V2,i);
V1.exchange(R, Section(1,100), Section(1,100));
V2.exchange(R, Section(1,100), Section(1,100));

for(i.begin(), i1.begin(), i2.begin(); i.end();
    i.next(), i1.next(), i2.next())
   R[i] = V1[i1] + V2[i2];
```

Distribution The distribution that we consider is called a *user-defined partitioning* since it lets the users define their own distribution. Given a two-dimensional array A, the distribution will correspond to a 3-tuple (i, j, p). It means that $A(i, j)$ is different from zero and it is distributed to the processor p. This description is considered as an interface between the distribution and its internal representation.

3 Descriptor

In PARADEIS, the description of the distribution is based on a conservative approximation because the information usually can't be replicated in each processor due to the amount of information which cannot be held in a processor memory. Hence, the approximation reduces the amount of information. A valid approximation is such that, given a non-zero value located at the coordinates (i, j), if the value at (i, j) is distributed to p then the approximation also gives this information. This description will guarantee that a communication requires only one message [4]. No supplementary message is necessary to find the location of a data. A *sparse tree descriptor* describes this approximation. According to this scheme, the mapping is described by a set of 3-tuples $([l_x : u_x], [l_y : u_y], p)$ signifying that the data items contained in this block are mapped to the processor p. Logically, the descriptor can be viewed as a tree. The descriptor shares some of its motivation (the approximation scheme for example) with the R-trees data structure frequently used in geographical database.
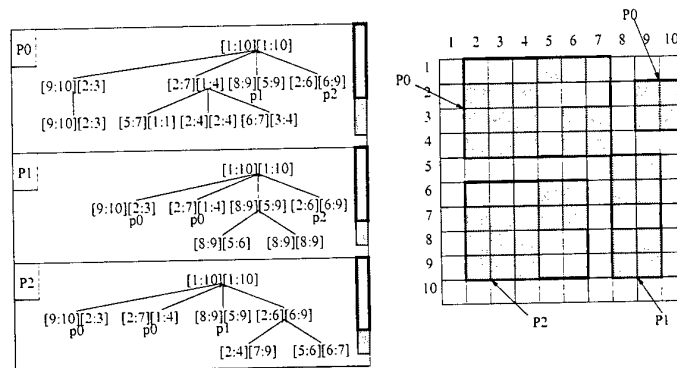
Fig. 1. Example of a descriptor codification

- The root is the size of the matrix.
- The first level is the common global knowledge shared by every processor. It provides an approximation of the exact mapping
- The second level corresponds to local knowledge. It is the exact description of the values held by a processor. This information is only stored on the processor where the corresponding data are mapped.
- The leaves contain data if they are resident on the processor.

Figure 1 describes an example of this codification for a 10×10 matrix. Filled cells correspond to significant values whereas empty cells correspond to zeros.

Any partitioning can be applied providing it is described by a sparse array descriptor. For matrices, some partitioning algorithms such as BRD decomposition [2] provide a natural description of partition as a sparse tree descriptor. If the result cannot be directly codified by blocks, then it is decomposed into several independent blocks. These blocks are not necessarily contiguous but they still refer to a single partition. Once the mapping is defined the pieces of the sparse matrix are assigned to processors. The codification is performed in parallel. The part of the global view computed for one processor is broadcast to every processor.

4   Experiments

In this section, we present experimental results on PARADEIS. The results of execution time are performed on the matrix vector product which is the core of numerous numerical algebra algorithms. In this program, the matrix is sparse and the two vector are dense. Experiment were running on an 8 nodes IBM SP2 (Power PC 66.6 Mhz, 256 Mb/processor). The tests have been running on several Harwell Boeing colllection matrices. Only the more interesting results are presented here.

This experiments on PARADEIS are meant to evaluate the overhead in computational time and in memory occupancy. The overhead in runtime execution is divided into two part, a constant time on each processor and a varying time depending on the number of processors. This varying time depict the penalty of the parallel execution management. The scalability experiment gives the execution time on multiple processors. It allows to compute the ratio of computing over idle time thus giving a measure of the varying time. The constant time comes from a software layer managing the iterators. A comparison with an equivalent sequential program (here the CRS matrix vector product) gives a measure of this overhead. The memory overhead is compared to a sparse data storage (CRS) to show the impact of the parallel description and of the density ratio. This ratio is used to build the leaves of the data structure by giving the number of zero that may be stored with non zero. Reducing this ratio stores more zeros but it reduces the descriptor size and vice versa.

The main properties of some of the $N \times N$ tested matrices are shown in the Table 2. They are NZ the number of non zero, N the size of the matrix and its density (how much spase it is).

	bcsstk13	cavity09	mcfe	bcsstm13	bcsstk19
NZ	42943	32747	24382	11973	3835
N	2003	1182	765	2003	817
Density	1.07 %	2.34 %	4.16 %	0.3 %	0.57 %

Fig. 2. Properties of the tested matrices

Scalability Figure 3 shows scalability and correpsonding efficiency of the matrix vector product with PARADEIS. The minimum efficiency is 75 %, which shows that the management of the parallel descriptor has little impact on execution time. The decreasing of the scalability is created mainly by communications.
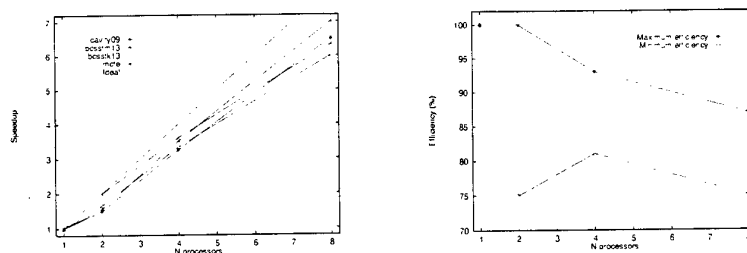
Fig. 3. Scalability and efficiency of the matrix vector product

Language Overhead In order to obtain the overhead in time induced by our iterator access and management, we evaluate the execution time of the matrix vector product writing in PARADEIS and the CRS storage version. From matrix to matrix, the ratio between execution time r is in the range $1.9 < r < 3$ on the RS6000 processor in favor of the CRS program. The loss compared to this specialized program comes from an increase of bound checking.

Memory Overhead The memory occupancy depends on two parameters: the chosen density ratio and the matrix structure. Figure 4 present two measures showing those two sides of the memory part in PARADEIS. The first one on the left shows the evolution of memory occupancy in bytes when the density ratio evolve. It is compared to a distributed version of the CRS storage (MRD Multiple Recursive Decomposition). This first measure show that memory occupancy is always close to the MRD and that its evolution with the density ratio depends on matrix structure. The second measure on the right present a detailed overview of memory occupancy distribution between the descriptor and the data. It shows that the global descriptor part is neglibible and that the local part depends on the matrix structure, growing faster when the values are more scattered.
This experiment shows that PARADEIS can be compared to a dedicated sparse matrix vector program and that it uses a moderated memory occupancy in the **SparseArray** data structure providing powerfull distribution and communication scheme. PARADEIS is a trade-off between expressiveness and efficiency for sparse linear algebra program.

5   Discussion and Conclusion

We consider PARADEIS as a compiler back-end as well as a user library. Thus, it provides some features to interact with the higher levels of a
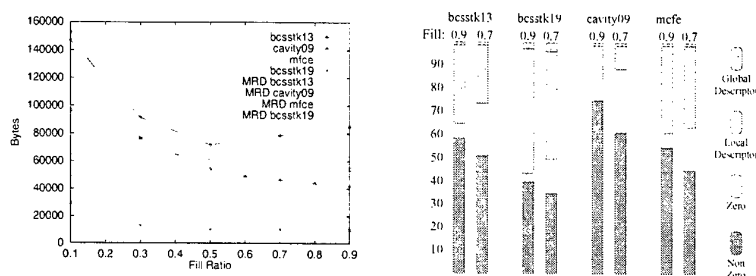
Fig. 4. Memory occupancy

sparse and/or a data-parallel compiler. PARADEIS addresses a class of sparse linear algebra problems with user-defined partitioning.

It is based on a run-time support which offers a scalable and portable framework for data-parallel programs which operate on sparse matrix. The portability is due to the language which has been installed in a large number of platforms since PARADEIS has been written in C++ (9000 lines) and the communication library is PVM. The scalability relies on the primitives. The conversion of global to local address is handled by methods and the collective exchange based on a global address space. It leads to a scalable framework since the user (or the compiler) has not to determine the exact contain of a message according to the amount of data folded in a processor. The mapping phase and the inspector-executor phase are distinguished. We think that it offers a modularity in the development since the both components can be improved or changed independently.

References

1. R. Adle. *Outils de parallélisation des programmes dense pour les structures creuses.* PhD thesis, University of Evry, 1999.
2. M.J. Berger and S.H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Comput.*, 36(5):570–580, 1987.
3. A.J.C Bik and H.A.G Wijshoff. Automatic data structure selection and transformation for sparse matrix computation. *IEEE Transactions on Parallel and Distributed Systems*, 7(2):1–19, 1996.
4. F. Delaplace and D. Remy. PARADEIS: An object library for parallel sparse array computation. In *ACPC99*, Salzburg,Austria, February 1999.
5. V. Kotlyar, K. Pingalli, and P. Stoghill. Compiling parallel code for sparse matrix applications. In *SuperComputing.* ACM/IEEE, November 1997.
6. Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

Installation routines for linear algebra libraries on LANs *

Domingo Giménez[1] and Ginés Carrillo[2]

[1] Departamento de Informática: Lenguajes y Sistemas Informáticos.
Univ de Murcia. Aptdo 4021. 30001 Murcia, Spain.
domingo@dif.um.es
[2] Departament de Tecnología
Universitat Pompeu Fabra.
La Rambla, 30-32. 08002 Barcelona, Spain.
gines.carrillo@tecn.upf.es

Abstract. In this paper we study the design of installation routines for linear algebra routines on networks of processors. The main idea is to develop those installation routines in such a way that they allow inexpert users to execute the parallel linear algebra routines with an optimum number of processors and distribution of data. The designing methodology of these routines has been analyzed for homogeneous and heterogeneous networks, and the experimental results obtained with a gaussian elimination routine are shown.

1 Introduction

In the last years parallel/distributed computing has become widely popular due in part to the possibility of using some processors connected by a communication network as a parallel system. In that way the use of parallel systems has become cheaper and easier, and new users are begining to use parallel programming. In particular, users with great computational necessities (scientists and engineers) now have the possibility of solving their problems using the machines they have access to, connecting them by means of some communication network (ethernet, fastethernet, myrinet, ...) and using message-passing parallelism, without high additional cost. But the design of message-passing parallel programs is not an easy task, specially for inexpert users. The problems these users need to solve are in many cases linear algebra problems: solution of linear systems, or eigenproblems. This is why we are working on the design of linear algebra routines especially for LANs (Local Area Networks) [1].

There is other research in which the design of linear algebra routines for heterogeneous networks of processors is analysed [2-4]. In some cases the distribution of data in the system is obtained dynamically [2], and statically in others [3]. Our goal is to statically obtain data distributions close to the optimum.

Since the users we are thinking of are not experts in parallel programming, one possibility is to develop installation routines for the linear algebra routines, in such a way that the user can execute a linear algebra routine in a processor. This routine consults information generated by the installation routine and decides the number of processors to use and the best data distribution to obtain the lowest execution time.

Each linear algebra routine has an associated installation routine which obtains approximate optimum values of the number of processors and the block size in which the matrices are divided. The installation routines are executed during the installation of the linear algebra library, but they can be re-executed each time the conditions of the system change, i. e. more memory in some machines, modifications in the management of the file system or addition or elimination of a processor.

As the system can be formed by processors with different capacities, it may be preferable to develop linear algebra routines and the corresponding installation routines for heterogeneous systems.

In this paper we analyse the methodology for the design of these installation routines for homogeneous and heterogeneous LANs. Experimental results obtained with a gaussian elimination routine and with variations in the heterogeneity of the network are shown.

2   Installation routines

A gaussian elimination routine has been used to study the methodology of the design of installation routines and the behaviour of these routines in systems with different characteristics.

The matrix is considered as divided in blocks of adjacent rows, and the blocks are assigned to the processors using a rowwise block-cyclic-striped mapping [5].

If the system is homogeneous the blocks can be all the same size (b). When the block size increases the imbalance increases, but the number of communications decreases. Thus, for a given matrix size (n), the installation routine must obtain the number of processors (p) and the block size, with which the lowest execution time is obtained.

In the case of a heterogeneous system, the block size is not the same for the different processors, and the blocks assigned to processors with higher computational capacity would be larger. The installation routine also obtains the optimum number of processors and block size, but in the linear algebra routine the size of the blocks in each processor is obtained by the formula:

$$b_i = \frac{v_i}{\sum_{i=1}^{p} v_i} pb \tag{1}$$

where v_i is the speed of processor i (i. e. in Mflops) and b_i the size of blocks assigned to processor i. This is the way in which other software for heterogeneous computing works [2] when dynamically deciding the data distribution in the system. In our case the assignment is not done dynamically since to dynamically

obtain the optimum number of processors and block size would be too costly. What could be done in our case is to dynamically obtain the values of v_i by executing in each processor some basic matrix operation (i. e. a matrix multiplication), but we have preferred to leave this work to the installation routine because we consider LAN systems are normally well controlled. In any case, the modification to obtain the block sizes b_i dinamically is only a small one, and it could be done in the linear algebra routine.

The installation routine could work by obtaining the values of p and b from a formula of the theoretical execution time (depending on the linear algebra routine) and the values of the cost of an arithmetic operation, of the start-up time and word-sending time obtained experimentally for the system. Some problems remain: to obtain the theoretical execution time we normally make some assumptions which are valid when the matrix size increases, but not with small matrices, and it is more difficult to predict the experimental results in LANs than in multicomputers, due to the characteristics of the communication network.

Another possibility is to obtain the optimum number of processors and block size by performing a number of executions. In that case the system manager decides the minimum and maximum matrix size and the increment of the matrix size. Experiments are performed for these matrix sizes, but to perform the experiments for all the possible numbers of processors and block sizes could be too expensive, and what the routine does is to obtain the number of processors and block size for the smallest matrix and it uses the values obtained for a matrix size as initial values for the next matrix size. Since the optimum values of p and b vary in a continuous manner with the matrix size, the cost of the installation routine thus becomes acceptable.

In the homogeneous case, a file with the matrix sizes and the associated number of processors and block size is generated, and the linear algebra routine consults this file for the entry with the matrix size nearest to the actual matrix size in order to decide the number of processors and the distribution of the matrix in the execution.

In the heterogeneous case, that file is also generated along with an additional file with the proportional speeds of the processors, classified from fastest to slowest. The linear algebra routine takes the number of processors from the first file, the processors to use in the execution from the second file, and the block sizes are obtained using formula 1 with the value b obtained from the first file and the values v_i from the second file.

3   Experimental results

In this section the experimental results obtained using different installation routines for a gaussian elimination routine are shown. The entries of the matrices have been randomly generated, and a network of SUN Ultra workstations connected by ethernet has been used. Three networks are considered:

- A network of five SUN Ultra 1 with the same computational capacity, but with one of them managing the file system, and consequently this machine works more slowly. This is the more homogeneous system of the three, and it will be called HOM.
- A network obtained by adding a SUN Ultra 5 to HOM, which is quicker than the other processors. This network will be called HIB.
- A network with three processors: the SUN Ultra 1 which manages the file system, another SUN Ultra 1, and the SUN Ultra 5. Since this network can be considered the more heterogeneous, it will be called HET.

In figures 1, 2 and 3 the results obtained with the different installation routines are compared with those obtained experimentally. The figures show the quotients of the execution time obtained with the optimum number of processors and block size given by the different installation routines and the optimum execution time obtained experimentally. Figure 1 shows the quotient for HOM, figure 2 for HIB, and figure 3 for HET. The installation routines used are: HOMEXP, the homogeneous-experimental routine; HETEXP, the heterogeneous-experimental; and THEOR, the theoretical routine. THEOR is used in HOM to obtain the theoretical optimum number of processors and block size to use in each processor and in HIB and HET the same number of processors is used, but the block sizes are different in the different processors.
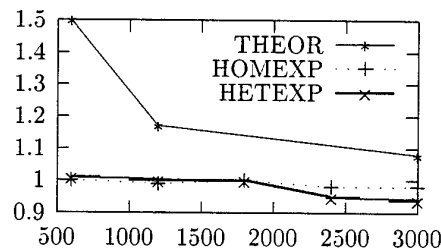
Fig. 1. Comparison between the different installation routines in HOM. Quotient of the execution time obtained with the values of p and b provided by the installation routines and the best execution time.

Some considerations can be made:

- The theoretical routine does not predict the number of processors and matrix distribution well for small matrices but when the matrix size increases the prediction is better. In some cases (HET) the theoretical prediction is as good as the experimental prediction. Theoretical prediction could be performed for big matrix sizes, since in that case the experimental routines are too expensive.

- The homogeneous-experimental routine works well when the network is close to a homogeneous network, but the heterogeneous-experimental routine is preferable when the heterogeneity increases.
- The heterogeneus-experimental routine works well in all the systems, so this type of routine is the best as installation routine.
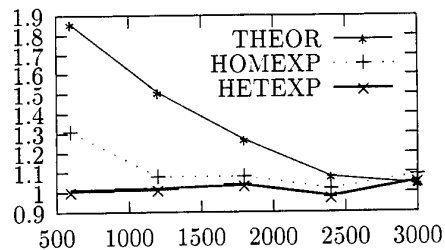
Fig. 2. Comparison between the different installation routines in HIB. Quotient of the execution time obtained with the values of p and b provided by the installation routines and the best execution time.
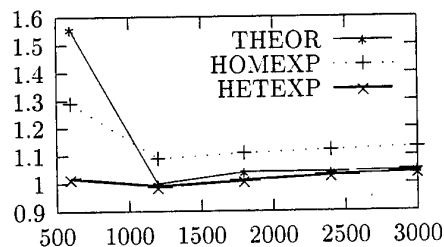
Fig. 3. Comparison between the different installation routines in HET. Quotient of the execution time obtained with the values of p and b provided by the installation routines and the best execution time.

4 Conclusions

We have shown a methodology to design installation routines for linear algebra libraries for non parallel programmers in LANs. These routines can be used to

provide the users with the number of processors and the block sizes for solving the problems in a time close to the optimum time.

The experiments performed show satisfactory outcomes.

Our idea is to design a linear algebra library using installation routines of the type analysed in this paper.

References

1. D. Giménez, C. Jiménez, M. J. Majado, N. Marín and A. Martín. Solving Eigenvalue Problems on Networks of Processors. In José M. L. M. Palma, Jack Dongarra and Vicente Hernández, editor, *Vector and Parallel Processing-VECPAR-98*, number 1573 in Lecture Notes in Computer Science, pages 85–99. Springer, 1999.
2. A. Kalinov and A. Lastovetsky. Heterogeneous Distribution of Computations While Solving Linear Algebra Problems on Networks of Heterogeneous Computers. In P. Sloot, M. Bubak, A. Hoekstra and B. Hertzberger, editor, *High Performance Computing and Networking*, number 1593 in Lecture Notes in Computer Science, pages 191–200. Springer, 1999.
3. Pierre Boulet, Jack Dongarra, Fabrice Rastello, Yves Robert and Frédéric Vivien. Algorithmic issues on heterogeneous computing platforms. *Parallel Processing Letters*, 9(2):197–213, 1999.
4. Vincent Boudet, Antoine Petitet, Fabrice Rastello and Yves Robert. Data Allocation Strategies for Dense Linear Algebra Kernels on Heterogeneous Two-dimensional Grids. In *IASTED Parallel and Distributed Computing and Systems*, 1999.
5. Vipin Kumar, Ananth Grama, Anshul Gupta and George Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms.* The Benjamin Cummings Publishing Company, 1994.

Some Remarks about Functional Equivalence of Bilateral Linear Cellular Arrays and Cellular Arrays with Arbitrary Unilateral Connection Graph

V. Varshavsky[1] and V. Marakhovsky[2]

[1] Neural Networks Technologies Ltd., Tel-Aviv, Israel
victor@nntlabs.com
[2] The University of Aizu, Aizu-Wakamatsu City, 965-8580 Japan
marak@u-aizu.ac.jp

Abstract. The considered task is building a cellular automaton, such that an array from automata of this type with arbitrary unilateral bivalent connection graph can solve the same problem as a bilateral linear cellular automata array. It is presumed that the complexity of the cellular automaton does not depend on the number of the automata in the array and, maybe, depends in some regular way on the rank of the respective graph vertex.

1 Introduction

First of all, we will try to give a more or less precise definition of functional equivalence as we treat it in this paper.[1] Of course, this definition directly depends on the way we treat *function* realized by the array, *problem* solved by the array and *behavior* of the array. Different treatment of these terms will lead to different results and different interpretation of them. When speaking of *problem* solved by the array, we will follow the classical examples from the works by Hennie [1], Fisher [2], Myhill [3], etc. We are going to deal with problems for which the formulation is invariant to the array size (number of automata), i.e. the automaton complexity (number of internal states) does not depend on the array size. The typical examples of such problems for one-dimensional arrays (chains of automata) are calculation of symmetrical Boolean function[4, 5], multi-valued voting problem[5-8] and firing squad synchronization problem[5, 7-15], that are formulated as follows[2].

Calculation of symmetrical Boolean functions [4, 5]. There is a bilateral linear automata array of $n + 1$ identical automata (Fig.1,a). Every automaton has two external inputs fed by Boolean function variables x_j and variables r_i determining

---

[1] It is strange, but this definition turned out to be the most complicated thing for us, already after successfully solving the problem of building an equivalent array.

[2] Solving these problems is beyond our scope. Our goal is showing by examples what sort of problems can an array solve.
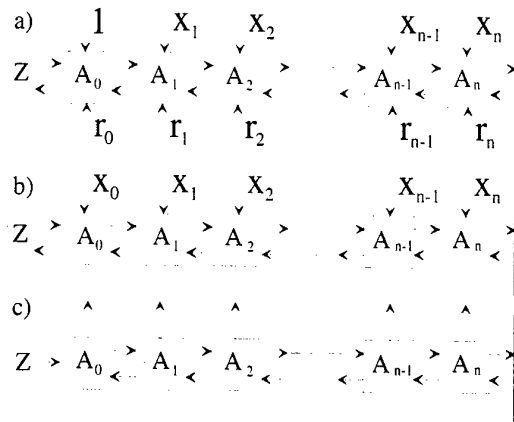
Fig. 1. Automata arrays calculating symmetrical Boolean functions (a), solving the multi-valued voting problem (b), and solving firing squad synchronization problem (c).

the working numbers of a symmetrical Boolean function (if $r_i = 1$, then the function contains the i-th working number). The lateral inputs (outputs) are fed by the states of the right and left neighbors (the automaton proper current state)[3]. The input of the Z junction is fed by the initiation signal. After some time the automaton A_0 must go to the state respective to the function value.

Multi-valued voting problem [5-8]. The problem itself is the following. There are N k-valued variables $X_i = \{0, 1, 2, ..., k - 1\}$, and m_j is the number of variables that take the value j. The multi-valued voting function is

$$F(X_0, X_1, ..., X_{n-1}) = a \quad if \quad m_a = \max_j(m_j). \tag{1}$$

From (1) the problem formulation comes for a bilateral linear-homogeneous automata array. The external inputs of the automata (Fig.1,b) are fed with k-valued external variables. Some time after the initiation signal arrives at Z junction, A_0 automaton must hit to the state respective to the value of the multi-valued voting function.

Firing squad synchronization problem [5, 7-15]. In this problem, the automata do not have the external informational inputs (Fig.1,c). After the input Z is fed by the external signal, all the automata have to simultaneously go to a final state after some delay under the condition that non of them does not hit to this state before the moment of common synchronization.[4]

Other problems also can be formulated. However, these three examples are enough for the goals of our article. They cover three classes of linear arrays: arrays

---

[3] For arrays from Moore automata.

[4] This problem has been also formulated and solved for the case of initiating an arbitrary automaton in the chain [5,7,8,14].

with homogeneous external inputs (the enumeration of the external inputs is not related to the automata enumeration — *multi-valued voting problem*), arrays with non-homogeneous external inputs (the enumeration of variables r_j is related to the automata enumeration — *calculation of symmetrical Boolean functions*) and arrays without external inputs (*firing squad synchronization problem*).

We deliberately do not consider here solving these problems for minimum time or by automata with minimum number of states; we are interested only in solution existence. In the same way, when discussing the functional equivalence problem, we will be interested only in principal possibility of building an automaton for arbitrary unilateral array from a known automaton for linear bilateral array, providing the solution of a similar problem.

2 Modeling the behavior of linear bilateral arrays by unilateral rings [5]

Let us consider a bilateral chain of Moore automata without external inputs. The lateral output of automaton A_j at the moment $t + 1$ is its internal state $a_j(t + 1)$ which depends on the states of its neighbors at the moment t:

$$a_j(t + 1) = f(a_{j-1}(t), a_j(t), a_{j+1}(t)), \quad 0 \leq j \leq n \tag{2}$$

where a_{-1} and a_n are the boundary conditions. First, instead of a bilateral chain of Moore automata let us consider a bilateral ring of Moore automata where the boundary conditions are replaced by modulo-n adjacent indexes. Let us put in correspondence to such ring a unilateral ring of n automata B_j (Fig.2) where the j index is counted by $\mod(n)$. Every automaton B_j is a composition of two sub-
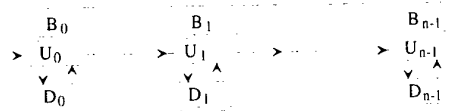
Fig. 2. Unilateral ring of automata.

automata U_j and D_j. Let us define the transition functions of these automata differently in even and odd time moments:

$$d_j(2t) = u_j(2t - 1), \quad u_j(2t) = u_{j-1}(2t - 1). \tag{3}$$

$$d_j(2t + 1) = u_j(2t), \quad u_j(2t + 1) = \varphi(u_{j-1}(2t), u_j(2t), d_j(2t)). \tag{4}$$

By substitution (3) into (4) we obtain

$$u_j(2t + 1) = \varphi(u_{j-2}(2t - 1), u_{j-1}(2t - 1), u_j(2t - 1)). \tag{5}$$

Let the automaton U have the same set of states as the automaton A has and the same transition function at time moments $2t + 1$. Then, taking into account

the shift of B automaton states by one ring position in every two cycles of its functioning, we can finally write:

$$u_{(j+t) \bmod n}(2t+1) = a_j(t). \tag{6}$$

The equation (6) provides the way in which we understand the statement that the unilateral ring from automata B simulates the behavior of the bilateral ring from automata A.[5] Note that the sub-automaton D_j is just an extra memory register on which at even moments the state of U_j is stored.

Let us now go back to the bilateral linear array. It differs from a bilateral automata ring only by the presence of the boundary conditions that make the internal and edge automata different. Since in the unilateral automata ring the initial automata enumeration during the functioning is shifting along the ring, the information about the boundary conditions also must be shifting along the ring with the same speed. This can be provided in several ways. For example, by introduction into the automaton B_j the sub-automaton Y_j with two states ($y_{n-1} = 1$, $y_j = 0$, $j \neq n - 1$). Then the equations (3) and (4) look like following:

$$d_j(2t) = u_j(2t-1); \quad u_j(2t) = u_{j-1}(2t-1); \quad y_j(2t) = y_{j-1}(2t-1);$$
$$d_j(2t+1) = d_j(2t); \quad y_j(2t+1) = y_j(2t); \tag{7}$$
$$u_j(2t+1) = \varphi(u_{j-1}(2t), u_j(2t), d_j(2t), y_{j-1}(2t), y_j(2t)).$$

The value combinations of variables $y_{j-1}(2t), y_j(2t)$ mean: 00 — internal automaton; 10 — extreme left automaton; 01 — extreme right automaton. In the problems discussed above, the extreme left automaton is fed by the initiating signal and the same automaton generates the output signal.

In the same way, the external variables X_j are incorporated that also must be shifting along the ring together with the working indexes of the automata. The variables are stored in the registers and $X_j(2t) = X_{j-1}(2t-1)$.

The above is enough to build a unilateral automata ring by an automaton that provides bilateral automata chain solution of, at least, problems of the types we mentioned in the introduction.

3 Modeling the behavior of a unilateral ring by an automata array with arbitrary connected unilateral bivalent graph of connection

Like in the previous section, we will treat the "possibility of modeling" as the existence of an automaton whose number of states and transition function does not depend on the number of vertexes and on the connection graph of the modeling array. Besides, this automaton must regularly depend on the valence of the vertexes and can be built by the automaton of the modeled ring.

In order to reduce the problem to the one we have already solved, let us first ask ourselves whether an arbitrary unilateral graph can be re-commutated as a

---

[5] This result was published in 1973 in a Russian journal [5]; it is practically unknown among the specialists.

ring. If it can, the problem is principally solved. The base for the positive answer to this question can be the existence of Euler cycle in the graph. The conditions for the existence of Euler cycle in the graph are the connectness and bivalence of the graph [16-18].

Let to every vertex of the graph having k incoming and k outcoming arcs (property of bivalence) put into accordance a composition of two automata, functional and control ones, and a fully accessible commutator $k \times k$ (Fig.3). The automaton compositions that correspond to vertexes of different degree will differ only in the commutator size. Talking about the commutator, we will
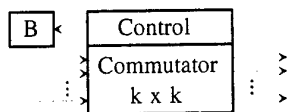
Fig. 3. Structure of the automaton, corresponding to to a graph vertex with k incoming and k outcoming arcs.

assume that the arcs are bundles of wires connecting the vertexes (automata). Via these wires the automaton sends the information about its state and receives the information about the state of some other automaton that is found by the commutation algorithm to be the succeeder of this automaton in the ring. Note that only one of input arcs is connected to the corresponding commutator input through the functional automaton B_k. If the direct connection of the input l_{ik} and output l_{kj} arcs of the vertex a_k is considered as electrical wire connection, then the procedure of such commutation[6] is equivalent to removing the arcs l_{ik} and l_{kj} from the graph and creating a new arc l_{ij}. Hence, if we build, as a result of the commutation, an Euler cycle from an arbitrary connected bivalent unilateral graph with n vertexes, this would be functionally equivalent to a unilateral ring of n automata.

Let us consider the behavior of the composition of controlling automata and commutator in solving the problem of building the Euler cycle. First, we should base upon some algorithm that would provide finding the Euler cycle for an arbitrary unilateral connected bivalent graph. The simplest algorithm of this type could seemingly be the following: "when going through an arc, mark it; when leaving a vertex, follow the unmarked arc". However, as shown by Ore, using this algorithm leads to Euler cycle only for a very limited subclass of unilateral connected bivalent graphs [19]. We will use here the algorithm by Hoang Tuy [17] as it was formulated by Zykov [18], but with some modification discussed below.

At the initial moment, all the commutators are disconnected, i.e. no one input arc is connected to any output arc. All the control automata are in the passive state P. The initiating external signal $X = 1$ arrives at a certain automaton

---

[6] In case when there is no functional automaton between these arcs.

A_0 drawing it to the working state M_1.[7] The state (token) M_1 goes to the first output arc (the enumeration is arbitrary). Control automaton A_j receives the token M_1 via one of its input arcs and goes to the state M_1, producing a signal for the commutator. The latter commutates the input arc via which the token has come with the first output arc, that means passing this token to the next vertex. This procedure will last until the token M_1 appears again on one of the input arcs of some vertex. In this situation, the automaton keeps its internal state M_1, commutating the input arc via which the new signal has come with the next free output arc. This process continues until it turns out that the commutator does not have any free output arc. Since the graph is bivalent[8], this can occur only in the initial vertex. Indeed, up to this moment we have been following the algorithm "when coming to a vertex, go to the first non-passed arc." In this case, as Ore [19] showed, no one arc will be passed twice; however, in an arbitrary graph non-passed arcs and vertexes may remain. In the next phase, we will pass some vertexes for the second time. The commutator of the initial vertex commutates the last initiated input arc with the first output arc via which the first signal M_1 was sent, creating a cycle. The control automaton goes to the state M_2, injecting the respective signal (token M_2) into the cycle. Token M_2 is passed to the adjacent automaton. This automaton commutator can be in one of the following two states:

– All the input arcs are commutated with all the output arcs. In this case, the control automaton goes to the state M_2, translating the token M_2 to the output.
– The commutator has at least one output arc. In this case, the control automaton keeps the state M_1, breaks the commutation of the input arc via which the token has come and commutates this arc with the next free output arc, injecting there the token M_1 (Fig.4).
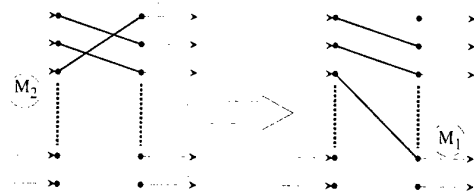
Fig. 4. Commutation in the case of coming the token M_2 by one of the input arcs of a vertex when it has at least one free output arc.

Let us assume that the token came to a vertex J via its input arc l_{ij} commutated with output arc l_{jk}. Breaking this connection, we send the token M_1 via the output arc l_{jr} which had been free. The token M_1 gets to a certain

---

[7] This can be either a specially allotted automaton, for example, an automaton that models the left lateral automaton in the linear array, or an arbitrary automaton.
[8] Every vertex has an equal number of input and output arcs. This number can be different for different vertexes.

connected bivalent subgraph G_J formed by vertex J and some subset of its non-passed (non-commutated) arcs. Since this subgraph is connected and bivalent. after some time M_1 will come back to the vertex J via a certain arc l_{mj}. Like in the previous case (subgraph associated with the initial vertex). M_1 will come back to the vertex J without hitting to already commutated paths until only one non-commutated output arc, namely l_{jk} remains in the vertex J. The input arc via which M_1 has come is commutated with l_{jk} and the control automaton passes the respective token (M_1) to the output via l_{jk}. After that the marker M_1 propagates only within already commutated cycle until it comes back to the initial vertex where it turns to M_2 according to the rule described above, keeping the internal state of the control automaton.

Thus, every time the markers M_2, M_1 rotates, another path is added to the initial cycle. This continues until M_2 comes back to the initial vertex. It means that all the commutators on the way of M_2 have been really commutated. i.e. the Euler cycle has been completely formed.[9] Indeed, the fact that M_2 has come back to the initial vertex indicates that no one vertex it has passed has free (non-commutated) output arcs (otherwise, the token M_2 would be replaced by M_1), i.e. all the arcs of the graph have been passed by M_2 and they form a cycle.

The appearance of M_2 at the input of the initial vertex switches the control automaton to the working state W and passes the control to the functional automaton that prior to this time has only translated markers to its output. If a signal belonging to the set of functional automaton states appears at the input of any other vertex different from the initial one, the control automaton of this vertex goes to the state W and the control is passed to the functional automaton.

4   Conclusion

It follows from the above that there is a way of allowing by a bilateral chain automaton to construct an automata composition that would solve the same problem on a random unilateral bivalent graph. Doing so, we keep the basic property of the automata designed for this type of problems: the automaton complexity (number of internal states and transition functions) in every vertex of the graph does not depend on the number of vertexes and type of the graph. We are far from pretending that the suggested solution is optimal. either in terms of the algorithm complexity or in terms of the time needed for solving the problems. Our goal was just to prove that the solution exists, because this fact has been exposed some doubt in private talks and seminar discussions. Furthermore, because of this goal we did not continue our discussion to automaton construction, limiting ourselves by algorithm description.

On the other hand. it does not mean that any problem of those mentioned above can be solved by the method we suggest. For example, the problem of

---

[9] The difference of this algorithm from the Tuy's algorithm is that in the last initial cycle and added loops are enumerated by the sequence of numbers, but in our case there is no enumeration it is replaced by switching of two markers.

calculating symmetrical Boolean functions requires that the inputs are linked to the numbers of the automata in the ring, while the suggested algorithm of building the Euler cycle does not allow us to provide such a linkage.

Finally, let us note that it looks fairly interesting to consider the parallel algorithm of building the Euler cycle, when the initiating token is injected into the graph via all the output arcs of the initial vertex.

References

1. Hennie F.S., *Iterative arrays of logical circuits.* N.Y., Wiley, 1961.
2. Fisher P.C., "Generation of primes by a one-dimensional real-time iterative array", *Journal of Association for Computing Machinery*, vol.12, 3, 1965.
3. Moor E.F., "The firing squad synchronization problem", *"Sequential machines" reading*, Mass-Palo Alto-London, Addison-Wesley Publ. Co., inc., 1964.
4. Varshavsky V., Marakhovsky V.; Peschansky V., "Invariant functions realization with linear homogeneous circuits," *Soviet Journal of Computer and System Science*, No. 4, 1969, pp. 75-78 (in Russian).
5. Varshavsky V., Marakhovsky V., Peschansky, V., Rosenblyum L., *Homogeneity Structure. Analysis. Synthesis. Behavior.*, Moscow, Energy, 1973 (in Russian).
6. Varshavsky V., Marakhovsky V.; Peschansky V., "About voting problem in an automata chain," *Soviet Journal of Computer and System Science.* No. 4, 1968, pp. 94-100 (in Russian).
7. Varshavsky V., "Collective behavior and control problems", *Machine Intelligence III*, Edinburgh University Press., 1969.
8. Varshavsky V., "The organization of interaction in collectives of automata", *Machine Intelligence IV*, Edinburgh University Press., 1968.
9. Goto E., "A minimal time solution of the firing squad problem", *Dittoed cource notes for applied mathematics 298*, Harvard University, 1962.
10. Moore E.F., "The firing squad synchronization problem", *Sequential machines reading*, Addison-Wesley Publ. Co.,Inc.,1964.
11. Levinshtain V., "About a decision method of the automata chain synchronization problem for minimum time", *Problems of Information Transmission*, vol. 1, no. 4, 1965 (in Russian).
12. Waksman A., "An optimum solution to the firing squad synchronization problem", *Inf. and Control*, vol. 9, 1, 1966.
13. Balzer R., "An 8-state minimum time solution to the firing squad synchronization problem", *Inf. and Control*, vol. 10, 1, 1967.
14. Varshavsky V., Marakhovsky V., Peschansky V., "Some variants of automata line synchronization problem", *Problems of Information Transmission*, vol. 4, no. 3, 1968, pp.73-83 (in Russian).
15. Varshavsky V., Marakhovsky V.; Peschansky V., "Synchronization of interacting automata", *Mathematical System Theory*, vol. 4, no. 3, pp. 212-230, 1970.
16. Euler l., "Solutio problematis ad geometriam situs pertinentus", *Commentarii Acad. Petropolitanae*, 8, 1736, pp.128-140.
17. Hoàng Tuy, *Dô thi hü'u han và càc ù'ng dung trong vân trù hoc*, Nhà Xuâ't Bàn Khoa Noc, 1964,142p.
18. Zykov A.A., *Fundamentals of Graph Theory* BCS Associates, 1990, 371p., (ISBN: 0-914351-04-4).
19. Ore O., *Theory of Graphs*, Amer. Math. Soc. Colloq. Public., 28, 1962.

Preliminary Results of the PREORD Project:
A Parallel Object Oriented Platform for DMS Systems

M. Pedro Silva
msilva@lorde.inescn.pt

J. Tomé Saraiva[1]
jsaraiva@inescn.pt

Alexandre V. Sousa[2]
avs@ismai.pt

INESC Porto - Instituto de Engenharia de Sistemas e Computadores
Largo de Mompilher, 22, 4007 Porto Codex, Portugal
[1] FEUP/DEEC - Faculdade de Engenharia da Univ. do Porto
[2] ISMAI – Instituto Superior da Maia

Abstract - In this paper we describe a software package corresponding to a DMS – Distribution Management System – that is structured in terms of a distributed multitask client-server architecture. The system is implemented using Object Oriented technology and integrates a number of power system application tools that are structured in terms of main coordinator objects calling and directing the object models of system components as well as other auxiliary interface and calculation objects. These tools can be activated by several entities corresponding to clients in terms of a distributed architecture.

1. Problem Positioning

For several years electric utilities directed a major percentage of their investments to the generation and high voltage transmission systems. This fact explains that generation and transmission high voltage systems are now characterized by higher levels of automation and performance indices both in terms of economic efficiency and reliability levels. The referred trend started to change some years from now with the consequence that today much more of the efforts are directed to the distribution area. These efforts lead to the development and installation of new automation, telemetering and communication facilities at the distribution level in order to have tools to monitor the networks, to operate systems in a remote and central way and to reduce the number of interruptions as well as the interruption times. The investments in the distribution area, together with new technological advances, made it feasible to have in real time in Control Centers values for an increasing number of variables as well as indications regarding the topology in operation.

Anyway, distribution networks have some distinctive aspects when compared with higher voltage transmission systems that prevent the direct migration of solutions and applications common and well established in EMS – Energy Management Systems – at the generation/transmission level. Apart from that, the size of distribution networks turns it most probable that the investments will be distributed along a large number of years until an adequate level of automation and tele-operation is achieved. Finally, the presence of large numbers of independent generators and the liberalization of the electricity sector is already imposing non negligible impacts in the distribution sector as the eligibility levels for accessing the open markets start to decrease.

All these changes and challenges suggest it is crucial to develop new DMS – Distribution Management Systems – according to the requirements of the distribution

sector in the advent of the open market. This means we would not be tied to solutions already developed in the generation/transmission systems and integrated in EMS systems, but we would rather be concerned in developing a new system. This motivated our team to work on this area and to present to the PRAXIS program – Portuguese state program for financing scientific and development projects – the PREORD proposal in order to develop a new DMS system using new technologies and more advanced programming facilities. In any case, the development of DMS share with EMS some general guidelines (see reference 6. for instance) as flexibility, reusability, openness (in terms of their portability, interoperability, interconnectivity and scalability), security and accessibility.

The software package described in this paper can be integrated in the move to an increasing level of automation of distribution systems. Apart from that, one witnesses an important evolution in several technological aspects regarding databases, programming languages and hardware structures themselves and new requirements in getting a more intuitive and friendly interface with the user and of supporting more complex and involving functions. In this scope, several applications were developed in recent years adopting the Object Oriented paradigm. As examples, references 1, 3, 4, 5, 8, 10 and 11 describe several applications of Object Oriented technology to power systems. In this scope, these references describe general power system models, graphical user interfaces, topology processor and power flow algorithms. According to these references, it is suggested that distributed systems in general, and Object Oriented applications in particular, are the most adequate and flexible approaches to be adopted to develop new generation DMS systems.

2. Object Oriented Basic Concepts

Under the Object Oriented paradigm, the objects correspond to the main units in the strategy adopted to solve a particular problem. The adoption of an Object Oriented approach mainly aims at catching the concepts of real world that are significant for the application being developed. Under this paradigm, real systems are usually structured in terms of a number of objects that can be grouped in Classes sharing a common set of variables and methods – eventually, calculation methods that manipulate the particular values assumed by those variables (see references 3. and 7.). The objects sharing the same information, from these two points of view, are included in a Class so that a particular object can be seen as an instance of that Class. From this point of view, variables in a particular object in a Class are assigned particular values corresponding to instances defined for those variables.

It is important to notice that the above Class definition is flexible enough to allow the construction of a hierarchical structure in which some Sub-Classes are defined under a Class placed at a superior level. In this structure, there is a common set of information – core – that is common to all Sub-Classes. This information, both in terms of variables and methods, is included in the Class at the superior level and it is inherited by all Sub-Classes. This organizational approach requires an higher level of abstraction in the sense that we will have to structure the system under analysis independently of particular instances of variables and recognizing what is common to

several objects – leading to a Class definition – and what is different between them – leading to Sub-Classes. Apart from this higher abstraction degree, the modularity is also favoured because a change in a private method or function at a certain level of the hierarchy does not affect the remaining Classes and Sub-Classes.

Regarding the DMS under development, we will describe in the next section its general architecture, the referred hierarchical structure as well as the Class Models for some particular objects corresponding to network devices and application functions.

3. The PREORD Platform

The software package has a modular multitask client-server architecture and is supported by a commercial Object-Oriented Database Management System – ObjectStore. The modules corresponding to specific algorithms and applications of the DMS are connected to this platform. The clients correspond to Java applets and the modules related to DMS applications or algorithms are developed in Java or C++ and are registered in the Java DMS server as services. The server is a Java application that uses the Java language facilities to handle concurrency in a transparent manner. The reliance on Java gives us platform independence. When a user opens an HTML page in the web server – the page contains a reference to the Java applet – the applet is downloaded and starts running in the machine of the client.

In our software package we use an Object Oriented approach as a way to build a mathematical model for the physical system to be analysed. The structural unit of this model is the object and they represent concepts existing in the real world. Each structural unit has a static identity and a dynamic associated to the transformations that can affect its state variables. The rules directing the interactions between different units of the system are also defined. The objects are grouped in classes so that it is possible to study the interdependencies inside the software in order to minimize them. We used a CASE tool to support the development process and generate the UML – Unified Modelling Language – diagrams (see reference 9.). The source code is under version control, and through the use of the CASE tool it is possible to automatically generate code from the detailed UML class diagrams.

In this development phase, special attention is devoted to the object models of the components of power system networks as, for instance, lines, cables, transformers and generators. These models integrate data corresponding to static and dynamic characteristics. Dynamic data are, for instance, voltage magnitude and phases, branch currents and generations. The object model of each component also integrates information regarding calculations that can be performed with static and dynamic information. Apart from that, the OO model of the system includes a number of objects to coordinate the actions of those component objects and other calculation or interface objects. From this point of view, each power application function – as for instance the power flow application - is structured in terms of a main coordinator object that has several sub-coordinator objects depending from it and that gives orders to component objects and other auxiliary calculation or interface objects.

The Object Oriented model of the system is organized in three levels:
- Electric Level – at this level the system has information about power system components. At the higher hierarchical position, there is the Electric Network that integrates information regarding Electric Connections, Electric Equipment and Ground. The Electric Equipment Object is organized in terms of a number of subclasses corresponding to one-terminal, two-terminal and three-terminal devices as detailed in Figure 1. The Electric Connection corresponds to a bus but, in order to give more flexibility to the software, we also included the sub-class OtherNetwork to represent the equivalent circuit of networks to which the network under analysis is connected. Finally, the Ground sub-class includes information regarding the connections of an electric network to the ground;
- Topologic Level – the available information regarding the models of electric devices, the connectivity and status of switching devices has to be analysed in order to produce a mathematical model of the system. This is achieved by Topology Processor application leading to a Class Diagram organized in terms of islands – energized or non-energized – corresponding to sets of electrically connected components (nodes and branches);
- Geographic Level – at this level the software will be interconnected with a Geographic Information System – GIS – given its ability to represent in different layers large amounts of data having a geographic dispersed nature;

According to Figure 1, the information regarding the components of power systems is structured in terms of one, two and three terminal devices. For illustration purposes, let us consider the Transformers. The information both for two and three winding transformers is structured in terms of the Sub-Classes Winding and Regulation. The Sub-Class Winding includes variables and methods designed to represent one winding of a transformer while Regulation includes information related to the voltage regulation abilities of a transformer. Finally, the Class 2Winding Transformer gathers all this information in order to model such a device (Figure 2).
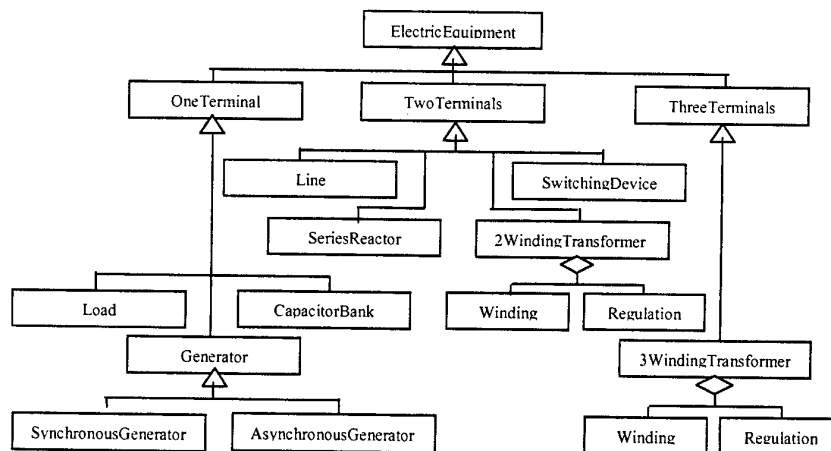
Fig. 1 – Class diagram for Electric Equipment.

2WindingTransformer
SnumberObjectsClass: int=0 -id: int -yshunt: Complex
+getId():int +getYshunt():Complex +getYShuntPrimary():Complex +getYShuntSecondary():Complex +getZSeries():Complex +getPrimaryWinding():Winding +getSecondaryWinding():Winding -newId():void -setYShunt(Complex newYShunt):void -computeYShuntPrimary():Complex -computeYShuntSecondary():Complex -computeZSeries():Complex +setPrimaryWinding(newWinding PrimaryWinding) +setSecondaryWinding(newWinding SecondaryWinding) +2WindingTransformer(int newTerminal1, newTerminal2, Complex newYShunt)

Fig. 2 – Class diagram for 2 Winding Transformer.

The client interacts with the server in order to request services by activating some objects organized in terms of calculation or coordinator objects. As examples, we present in the following paragraphs the main structure corresponding to the Topology Processor and Single Phase Power Flow.

The Topology Processor builds a simplified connectivity model of the system taking into account the position of switching devices. The Class TopologyProcessor includes two subclasses leading to the single phase equivalent and to the positive sequence circuits. The single phase circuit is used in the single-phase power flow and state estimation and the positive sequence circuit is used in the three phase symmetric short circuit analysis. As an example, SinglePhaseTP directs the following objects:

- Buses – simplifies the network by identifying individual nodes that are connected by closed switching devices and joining them;
- CreateIsland – checks, step by step, the connectivity of all nodes remaining in the system in order to identify islands and to create data structures for them;
- ClassifyBuses – classifies the buses in the system as PV and PQ and selects a PV bus for reference in each island;
- IslandClassification – this object classifies the pre-identified islands in terms of being energized or not energized;
- TracingFunctions – perform facilities as Single Tracing, Multiple Tracing, Tracing Upstream, Tracing Downstream and Tracing to Ground;

The Single Phase Power Flow is based on the Newton Raphson method and it runs for an island of the system identified by the Topology Processor. SinglePhasePF gets the id's of the equipments in the selected island and directs calculation objects as:

- Initialization – initializes voltages and phases for all buses;
- BuildInvJacobean – using sparcity techniques builds and inverts the Jacobean matrix at an iteration of the algorithm;
- EvaluatePQMismatches – computes injected powers and evaluates mismatches for active and reactive powers;

- <u>EvaluateVθIncrements</u> – computes increments of voltages and phases using the inverted Jacobean and the power mismatches and evaluates the convergence;
- <u>AdjustTaps</u> – checks if voltage taps have to be adjusted for on line voltage regulation transformers;
- <u>BuildResults</u> – computes the final values for voltages, phases, generated powers, power and current flows and losses.

4. Conclusions

In this paper we describe the main guidelines of a DMS software package that adopts a distributed client-server architecture. The DMS applications are organized in terms of services that can be activated by clients when entering in the DMS Web page. The distribution network is modelled using Objected Oriented Concepts and is structured in three levels – Electric, Topologic and Geographic. The power system applications are implemented in terms of calculation or coordinator objects given that they can be used to direct the activation of other objects and the flow of information. From the experience gained so far we consider that the use of OO technology corresponds to a major decision given the influence it has in all remaining development steps. Currently, we are finishing the implementation of the coordinator and calculation objects related to some power system applications and addressing issues related to real time processing considering that in a system as this one dynamic information is received from remote units periodically. At the end of this project we aim at having ready a prototype of a DMS system in order to test it in closer to reality environments.

References

1. Foley, M., Bose, A., Mitchell, W., Faustini, A., "An Object Based Graphical User Interface for Power Systems", IEEE/PES 1992 Winter Meeting, New York, Jan. 1992.
2. Foley, M., Bose, A., "Object-Oriented On-Line Network Analysis", IEEE Trans. Power Systems, vol. 10, no. 1, February 1995.
3. Fuerte-Esquivel, C., Acha, E., Tan, S. G., Rico, J. J., "Efficient Object Oriented Power Systems Software for the Analysis of Large-Scale Networks Containing FACTS-Controlled Branches", IEEE Trans. Power Systems. vol. 13, no. 2, May 1998.
4. Hakavik, B., Holen, A. T., "Power System Modeling and Sparse Matrix Operations Using Object-Oriented Programming", IEEE Trans. Power Systems, vol. 9, no. 2, May 1994.
5. Horiuchi, K., Hara, K., Horiiki, S., "Unified Object-Oriented Data Modeling on Electric Power Systems", in Proceedings of Stockholm Power Tech Conference, SPT'95, Stockholm, June 1995.
6. IEEE Task Force on Power System Control Center Database, "Critical Issues Affecting Power System Control Center Databases", IEEE Trans. on Power Systems, vol. 11, no. 2, May 1996.
7. Meyer, B., "Object-Oriented Software Construction", Prentice-Hall, 1988.
8. Neyer, A., Wu, F., Imhof, K., "Object Oriented Programming for Flexible Software: Example of a Load Flow"; IEEE Trans. Power Systems, vol. 5, no. 3, August 1990.
9. Rumbaugh, J., Jacobson, I., Booch, G., "The Unified Modeling Language Reference Manual", Addison Wesley 1999.
10. Zhou, E. Z., "Object-Oriented Programming, C++ and Power System Simulation", IEEE Trans. Power Systems, vol. 11, no. 1, February 1996.
11. Zhu, J., Lubkeman, D., "Object-Oriented Development of Software Systems for Power System Simulations", IEEE Trans. on Power Systems, vol. 12, no. 2, May 1997.

Dynamic Page Aggregation Technique for Nautilus DSM System - A Case Study

Mario Donato Marino and Geraldo Lino de Campos*

Department of Computer Engineering - Polytechnic School of University of So Paulo

Abstract. This paper introduces the dynamic aggregation of pages in Nautilus, which main features are: lock-based scope consistency, multi-threaded and page-based DSM system. The dynamic aggregation consists in considering a larger granularity's unit than a page, in a page-based DSM system. For the first time, an introductory evaluation of the influence of the dynamic aggregation technique in the speedup of a DSM with Nautilus's features is done. The first results show that this technique can improve the Nautilus's speedup up to 13.10%. The benchmarks evaluated in this study are SOR (from Rice University) and LU (from SPLASH-2).

1 Introduction

The evolution and the decreasement of costs of interconnection technologies and PCs have made the networks of workstations (NOWs) the most used as a parallel computer. Big projects such as Beowulf[11] can be mentioned to exemplify this.

The *Distributed Shared Memory* (DSM) paradigm[8], which has been largely discussed for the last 9 years, is an abstraction of shared memory which permits to view a network of workstations as a shared memory parallel computer.

In terms of granularity, DSMs have chosen in most cases page-grained approaches instead of fine-grained ones. Also, the study of Iftode[17] showed that for several applications from SPLASH-2, page-grain DSMs perform similarly to or better than fine-grain, although generally higher bandwith and message handling costs favor page-based DSM while lower latency favors fine-grained approach[17].

Some important DSMs which belong to the second generation like Quarks[7], TreadMarks[3], CVM[10], Brazos[18] and Nautilus[5], are page-based DSM systems. And, as it was said in the last paragraph, page-based solutions have achieved good speedups for several benchmarks, but there is still available place for improvements.

In page-based DSM systems, shared memory accesses are detected using virtual memory protection, thus one page is the unit of access detection and can be used as an unit of transfer. Depending on the memory consistency model and the situation, also the diffs[1] are used as an unit of transfer. For example, in homeless lazy release consistency (LRC), as TreadMarks, if the node has a

---

* {mario,geraldo}@regulus.pcs.usp.br

[1] diffs: codification of the modifications suffered by a page during a critical section

dirty page, diffs are fetched from several nodes, when an invalid page is accessed. On the other hand, in JIAJIA, pages are fetched from the home nodes when a remote page fault occurs.

The unit of access detection and the unit of transfer can be increased by using a multiple of the hardware page size. In this way, if aggregation is done, false sharing is increased. Aggregation reduces the number of messages exchanged. If a processor accesses several pages successively, a single page fault request and reply can be enough, instead of multiple exchanges, which are usually required. A secondary benefit is the reduction of the number of page-faults. On the other hand, false sharing can increase the amount of data exchanged and the number of messages[16].

The main goal of this paper is to evaluate the page aggregation technique[16] in Nautilus DSM system. The page aggregation technique is evaluated in Nautilus with a PC's network, with a free operation system. The speedups of TreadMarks made it the main DSM used by the scientific community, as a reference of optimal speedups. The speedups related to TreadMarks performance are used only as an allusion of good performance and other study[22] have confronted TreadMarks versus Nautilus, thus the main goal is not to compare TreadMarks and Nautilus.

The evaluation comparison is done by applying different benchmarks: LU (kernel from SPLASH-2)[15] and SOR (from Rice University). The environment of the comparison is a 8PC's network interconnected by a fast-ethernet shared media. The operating system used in each PC is Linux (2.x). This study is a preliminary evaluation of this technique and only two aggregation sizes are used: 4kB (default) and 8kB.

2   Nautilus DSM

The main motivation of the new software DSM Nautilus is to develop a DSM with a simple consistency memory model, in order to provide good speedups, and also another one with a simpler user interface, totally compatible with TreadMarks and JIAJIA.

Nautilus is a page-based DSM, as TreadMarks and JIAJIA. In this scheme, pages are replicated through the several nodes of the net, allowing multiple reads and writes[8], thus improving speedups. By adopting the multiple writer protocols proposed by Carter[2], false sharing is reduced and good speedups can be achieved. The mechanism of coherence adopted is write invalidation[8], because several studies [2][3][4][12] show that this type of mechanism provides better speedups for general applications. Nautilus, as JIAJIA does, uses scope consistency model, which is implemented through a locked-based protocol[13].

The implementation of the lock-based protocol is done in Unix using the *mprotect()* primitive. With this primitive, pages can be in RO, INV or RW states, thus pages can have their states changed easily.

Let's summarize Nautilus features: i) scope consistency; ii) multiple writer protocols; iii) multi-threaded DSM: threads to minimize the switch context; iv)

no use of SIGIO signals(which notice the arrival of a network message); v) minimization of diffs creation; vi) primitives compatible with TreadMarks, Quarks and JIAJIA. Nautilus follows the lock-based protocol proposed by JIAJIA[12], because of its simplicity, thus minimizing the overheads. Based on this protocol, the pages can be in one of three states: Invalid(INV), Read-Only (RO) and Read-Write(RW). In addition, the home nodes of the pages always contain a valid page, and the diffs corresponding to the remote cached copies of the pages are sent to the home nodes. A list with the pages to be invalidated in the node is attached to the acquire lock message.

3   Page Aggregation

In terms of implementation, following the other DSMs directions, in Nautilus there is a handler responsible for request a page from a remote node when a segmentation fault occurs. When a page is accessed and it's in the INV state a SIGSEVG signal is generated and the respective handler, as it was said before, requests the page from the home node. When the page arrives the primitive *mprotect()* changes the state from INV to RO.

When the page is written, another SIGSEGV signal is generated and the primitive *mprotect()* changes the state of the page from RO to RW. After the generation of the diffs, also with the *mprotect()* primitive, pages go to RO state again. And, when the write-notices, indicating the pages are modified by other nodes, arrive, pages go to INV state (again with the use of *mprotect()* primitive).

The primitive *mprotect()* permits to consider a granularity multiple of a page, thus giving the same permission for a region multiple of a page. Thus, this fact gives the condition to modify more than one page at the same time, which is named page aggregation technique.

The study [16] says that if aggregation is done, false sharing is increased and aggregation reduces the number of messages exchanged. Also, processor accesses several pages successively, a single page fault request and reply can be enough, instead of multiple exchanges of requests and replies, which are usually required. The study [16] also shows that there is a reduction of the number of page-faults, but false sharing can increase the amount of data exchanged and the number of messages.

This study is an original contribution because the study [16] is applied with TreadMarks, which is a lazy release consistency homeless system, and this technique until the present was not applied in other scope consistency, multi-threaded and for Unix DSM, which are Nautilus's features.

By changing the page size default (4kB) to, for example, 8kB using *mprotect()* primitive in Nautilus, it's possible to evaluate the effects of the incremented size in page fault reduction in the speedups.

4   Experimental Platform and Result Analysis

The results reported here are collected on a 8 PC's network. Each node (PC) is equipped with a K6 - 233 MHz (AMD)processor, 64 MB of memory and a fast ethernet card (100 Mbits/s) . The nodes are interconnected with a hub. In order to measure the speedups, the network above was completely isolated from any other external networks. Each PC runs Linux Red Hat 6.0. The experiments are executed with no other user process.

In this study, **two** sizes are considered for page size: 4kB, which is the default (memory hardware) and 8kB, which is multiple of 4kB.

The test suite includes some programs: LU (from SPLASH-2[15]) and SOR (from Rice University). The data input size N used in the LU evaluation is **N=1024**. The data input size of red and black matrix used in SOR evaluation is **1728x1728**; the number of iterations for the SOR benchmark is **10** .

Before presenting the results and their analysis, it is necessary to emphasize that the execution time for number of nodes = 1 in all evaluated benchmarks is obtained from the sequential version of the benchmarks without any DSM primitive. So, the primitive used to allocate memory to obtain the sequential time (number of nodes = 1) is **malloc()**, default primitive of C programming.

In order to have an accurate, homogeneous and fair comparison, the same programs are executed using TreadMarks (version 1.0.3). **There are some constraints with TreadMarks version (1.0.3) used:** i) the applications were executed and the speedups measured using *Nautilus* running on up to **8 nodes;** ii)**bigger input sizes:** the shared memory size is limited in this version; iii)only time and speedups can be obtained from this version, thus it was not possible to obtain number of page faults and SIGSEGV signals.

Table 1 show some features and results of the benchmarks: sequential time (t(1)), 8-processor parallel run time(8), speedup (Sp), remote get page request counts (gp) and number of local SIGSEGV of Nautilus(SG). The sequential time t(1) was obtained from the sequential program without no DSM primitives and **malloc()** primitive. In order to evaluate the adaptive write detection speedup, remote get page request counts and the number of local SIGSEGVs of Nautilus are taken. For table 1, **Tmk** means TreadMarks, **N4k** means Nautilus using 4kB page size and **N8k** means Nautilus using 8kB page size.

For both benchmarks evaluated in this study, a big reduction of SIGSEGV signals can be observed from table1, by looking at SG rows. Also, it can be noticed from this table a reduction of the number of page fault requests. These two results were obviously hoped because, as the page size increases, more data is included inside a page and as an immediate consequence, less number of page faults and requests for pages are necessary.

For LU, a reduction of 2.2% is observed when the dynamic aggregation technique is applied. Although the number of SIGSEGVs and the number of get page requests decreases by 36.98% and 19.37% respectively, as can be observed from 1, the employment of dynamic aggregation technique changes the data distribution. This new data distribution change the home nodes, giving a distribution not so adequate as the initial (4kB), decreasing the speedups of Nautilus.

app	LU	SOR
t(1)	350.90	29.10
t(8).Tmk	54.45	8.66
t(8).N4k	54.32	7.66
t(8).N8k	55.52	6.54
Sp.Tmk	6.44	3.36
Sp.N4k	6.46	3.80
Sp.N8k	6.32	4.45
SG.N4k	7980	12425
SG.N8k	5029	7912
gp.N4k	1528	118
gp.N8k	1232	72

Table 1. table comparing N4k x N8k

For SOR, wich has good data distribution, the dynamic aggregation technique decreased the number of SIGSEGVS by 36.00%, and also the number of pages requested by 38.00%. These reductions justify the increasement of the speedups of 13.1%.

The goal of this paper is not to compare Nautilus with TreadMarks, as it was done in the study of Marino[22]. For Matmul, Nautilus outperforms TreadMarks by 18.6%; for SOR Nautilus(4k) outperforms TreadMarks by 13.09% and Nautilus(8k) outperforms TreadMarks by 32.44%.

5   Conclusion

In this paper the page aggregation technique for a DSM which has similar Nautilus's features was presented. For reference of optimal speedups, TreadMarks was employed to have a fair comparison.

It was seen that the page aggregation technique has improved Nautilus speedups in until 13.10% for SOR benchmark, reducing the number of page faults and the number of SIGSEGVs. For LU, the dynamic aggregation technique decreased the speedup possibly due to the changing of the home nodes.

In addition, the speedup of Nautilus was compared to TreadMarks, but not as the main goal of the paper.

In our future works other applications will be tested and other page sizes, for example, 16kB and 32kB, also will be evaluated.

References

1. Carter J. B., Khandekar D., Kamb L., *Distributed Shared Memory: Where We are and Where we Should Headed*, Computer Systems Laboratory, University of Utah, 1995.
2. Carter J. B., *Efficient Distributed Shared Memory Based on Multi-protocol Release Consistency*, PHD Thesis, Rice University, Houston, Texas, September, 1993.

3. Keleher P. , *Lazy Release Consistency for Distributed Shared Memory*, PHD Thesis, University of Rochester, Texas, Houston, January 1995.

4. Hu W., Shi W., Tang Z., *JIAJIA: An SVM System Based on a new Cache Coherence Protocol*, technical report no. 980001, Center of High Performance Computing , Institute of Computing Technology, Chinese Academy of Sciences, January, 1998.

5. Marino M. D.; Campos G. L.; *A Preliminary DSM Speedup Comparison: JIAJIA x Nautilus*, to be published at HPCS99.

6. Li K, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, PHD Thesis, Yale University, 1986.

7. Swanson M., Stoller L., Carter J., *Making Distributed Shared Memory Simple, Yet Efficient*, Computer Systems Laboratory, University of Utah, technical report , 1998.

8. Stum M. , Zhou S. , *Algorithms Implementing Distributed Shared Memory*, University of Toronto, IEEE Computer v.23 , n.5 , pp.54-64 , May 1990.

9. Bershad B. N. , Zekauskas M. J. , SawDon W. A. , *The Midway Distributed Shared Memory System* , COMPCOM 1993.

10. Keleher P., *The Relative Importance of Concurrent Writers and Weak Consistency Models*, in Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS-16), pp. 91-98, May 1996.

11. Becker D., Merkey P.; *Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs*, Proceedings, IEEE Aerospace, 1997.

12. Eskicioglu, M.S., Marsland T.A., Hu W, Shi W.; *Evaluation of the JIAJIA DSM System on High Performance Computer Architectures*, Proceeding of the Hawai'i International Conference on System Sciences, Maui, Hawaii, January, 1999.

13. Hu W. , Shi W., Tang Z.; *A lock-based cache coherence protocol for scope consistency*, Journal of Computer Science and Technology, 13(2):97-109, March, 1998.

14. Iftode L., Singh J.P., Li K; *Scope Consistency: A bridge between release consistency and entry consistency*. Proceedings of the 8th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA'96), pp. 277-287, June, 1996.

15. Woo S., Ohara M., Torrie E., Singh J.P., Gupta A.; *The SPLASH-2 programs: Characterization and methodological considerations. In Proceedings of the 22th Annual Symposium on Computer* Architecture, pages 24-36, June, 1995.

16. Amza C., Cox A. L., Dwarkadas S., Jin L. J., Rajamani K., Zwaenepoel W., *Adaptive Protocols for Software Distributed Shared Memory*, Proceedings of IEEE, Special Issue on Distributed Shared Memory, pp. 467-475, March 1999.

17. Iftode L., Singh J. P.; *Shared Virtual Memory: Progress and Challenges*; Proceedings of the IEEE, Vol 87, No. 3, March 1999, 1999.

18. Speight E., Bennett J. K., *Brazos: A third generation DSM system*, In Proceedings of the 1997 USENIX Windows/NT Workshop, pp. 95-106, August, 1997.

19. Keleher P., Update Protocols and Iterative Scientific Applications, In The 12th International Parallel Processing Symposium, March 1998.

20. Keleher P., *Update Protocols and Cluster-based Shared Memory*, In Computer Communications, 22(11), pp. 1045-1055, July 1999.

21. Scales D. J., Gharachorloo K., Thekkath C., Shasta: A Low Overhead, *Software Only Approach for Supporting Fine-Grain Shared Memory*, In proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 297-306, October, 1996.

22. Marino M. D., Campos G. L., Sato L. M.; *An Evaluation of the Speedup of Nautilus DSM System* to be published at IASTED PDCS99.

A Parallel Algorithm for the Simulation of Water Quality in Water Supply Networks[1]

J. M. Alonso, F. Alvarruiz, D. Guerrero, V. Hernández, P. A. Ruiz, A. M. Vidal

Departamento de Sistemas Informáticos y Computación,
Universidad Politécnica de Valencia,
Camino de Vera s/n, 46022, Valencia. Spain.
Tel. +34 963877356
e-mail: {jmalonso, fbermejo, dguerrer, vhernand, pruiz,
avidal}@dsic.upv.es

Abstract. A parallel water quality modelling algorithm is presented for tracking dissolved substances in water-distribution networks. The algorithm, based on a parallel version of the Discrete Volume Element Method, contains an initial stage in which the water network is divided into several parts by means of the *Multilevel Recursive Bisection* graph partitioning method. The algorithm has been implemented and tested on a cluster of PCs with the MPI system, achieving good performance as shown in the results included.

1. Introduction

Computer simulation of water networks by means of mathematical models is nowadays common practice in most water companies, being an indispensable tool for various purposes. In particular, computer simulation is used, among other objectives, to guarantee the supply of the required water flows with the adequate pressures, ensure the existence of water stores in case of necessity, comply with water quality requirements, reduce energetic costs for the network operation, or reduce leakage.

The computational tasks related to the analysis of water networks are getting increasingly complex, due to various factors. First, the size and level of detail of the network models is growing, as a consequence of the incorporation of data from GIS (Geographical Information Systems). Second, it is nowadays increasingly frequent to be concerned with complex optimization problems. In this context, it has become patent the need of more powerful computing resources, and hence the interest in the use of parallel computing.

Consequently, the objective of the HIPERWATER project (http://hiperttn.upv.es/hiperwater) was to introduce High Performance Computing in the simulation and optimization of water networks, using the power of computing clusters to speed-up those tasks. The project resulted in the development of a software demonstrator, based

on EPANET, a well known water network simulation package [5]. HIPERWATER tackles three different problems making use of HPCN solutions [3], [4]:

- *Hydraulic simulation.* The problem consists in obtaining the value of flows and pressures in the different network components. The equations modelling water networks are non-linear and therefore require an iterative solution.
- *Water quality simulation.* By solving this problem information is obtained about substance concentrations, water age analysis, or percentage of flow from a determined source.
- *Leakage minimization.* The objective is to minimise leakage by controlling pressures with a number of Pressure Reducing Valves (PRV). This is done by means of a Sequential Quadratic Programming algorithm.

This paper is devoted to the second problem presented above. There are various methods that can be used for water quality simulation. In particular, we will consider here the *Discrete Volume Element Method* (DVEM) [6], which will be described next.

2. The Discrete Volume Element Method

A water distribution network is viewed as a collection of links connected together at their endpoints called nodes. Links can be of different types: pipes, pumps or valves. The purpose of the water quality simulation is to track the fate of a dissolved substance flowing through the network over time. The magnitude and direction of water flow throughout the network over time is taken as input data, being the result of the hydraulic simulation problem. In particular, we consider the type of hydraulic simulation known as *extended period simulation*, which divides the simulation period in a sequence of time steps, and in each of them the flows and velocities in the links are assumed to be constant.

The DVEM is formulated assuming a one-dimensional transport model. Within each hydraulic time step, a shorter water quality time step is computed, and each pipe is divided into a number of volume segments (elements). Then, advance and reaction of the substance is simulated through the following phases (see Fig. 1):

- *Reaction.* The reaction of the substance to be measured is simulated in this phase, if the substance is reactive.
- *Transport into nodes.* The mass of substance and volume of water of the last segment of each pipe is accumulated into its connecting node. Then, new concentration of the substance on each node is computed.
- *Transport along links.* Mass is shifted from volume element k to $k+1$ of each link.
- *Transport out of nodes.* Mass is moved out of each node into the first volume element of all outgoing links.

This sequence of phases is repeated until the start of the next hydraulic time step. Then the water quality time step is recomputed, the links are resegmented, and

computation continues. The method is fully explicit, in the sense that it does not require the solution of equation systems.
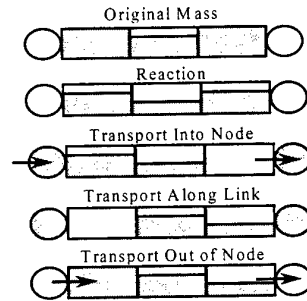
Fig. 1. DVEM phases for a link and its connecting nodes.

The water quality time steps used in the method are chosen to be as large as possible without causing the volume element size of any pipe to be larger than the volume of the pipe itself. Taking into account that the volume element size of a pipe is given by $Q_i\tau$, where Q_i is the flow in pipe i and τ is the water quality time step, τ must be chosen as

$$\tau = \min_i \frac{V_i}{Q_i}, \tag{1}$$

where V_i is the volume of pipe i. The quotient $\frac{V_i}{Q_i}$ is referred to as the *travel time* of pipe i. Then, the number of volume segments in each pipe is

$$n_i = \left\lfloor \frac{V_i}{Q_i\tau} \right\rfloor, \tag{2}$$

where $\lfloor x \rfloor$ represents the largest integer less than or equal to x.

3. Parallel DVEM algorithm

Different water quality time steps must be performed sequentially in time, due to the fact that the solution of a step requires the results of the previous one. Thus, the parallelization of the water quality process must be based on a parallel algorithm for each individual step.

In order to do so, we first divide the water network into several parts, one for each processor in our system. This initial network partitioning plays an important role to minimise communications and balance the computational load. Two are the desired objectives to be accomplished by the partitioning algorithm: a similar number of elements (nodes) should be assigned to each processor, and the number of pipes with nodes belonging to different processors should be minimum. The network can be

considered as a graph where the vertices are given by the nodes and the edges of the graph are the pipes and valves of the network.

In particular, the approach used is known as *Multilevel Recursive Bisection* technique [1], [2]. Since the partition of the network is carried out only once and it is not a time-consuming task (in the test networks the time involved is less than a quarter of a second), a serial version of this algorithm has been applied.

This algorithm works in the following way. First, a *coarsening phase* is performed, where the size of the graph to be partitioned is reduced, by collapsing vertices and edges. This reduction is repeated until a graph with a few hundred vertices is obtained. Then, in the *partitioning phase* a bisection of the small graph is carried out, and two subgraphs are obtained, with a minimum number of edges interconnecting them, and a similar amount of vertices in each subgraph. Finally, the *uncoarsening phase* takes place, where the objective is to project back the partition to the original graph, by means of a successive refining process.

This complete process leads to a good partition for the graph in a fast way. It must be noted that the graph partitioning determines how the nodes are assigned to each processor, but nothing is said about the distribution of the pipes. As one would expect, a pipe will belong to the processor owning their end nodes. If the two end nodes belong to different processors, the pipe will be arbitrarily assigned to any of them. Actually, this means that a frontier between network parts crosses nodes and not pipes, although the associated frontier in the graph crosses edges and not vertices. Whenever a graph frontier crosses an edge, the network frontier is moved to one of the two end nodes of the corresponding pipe. We refer to the nodes situated in a network frontier as *shared nodes*.

With the water network distributed among the processors, the parallel algorithm for the basic quality time step is largely given by the sequential one applied in each processor to the corresponding local portion of the network. Of course, some extra communication operations will have to be carried out, since the different network portions are not independent of each other. In particular, in order to perform the phase of "transport into nodes" for shared nodes, each processor has a local instance of these nodes into which the transport is done, obtaining the local values of mass and volume. After this phase, a communication operation is required in which the local contributions of the shared nodes are combined to obtain the final mass and water volumes, values which are then sent back to the processors sharing the nodes (this communication operation is implemented by means of the MPI function `MPI_Allreduce`). The rest of the phases in the sequential DVEM algorithm are not altered.

On the other hand, the process of computing the water quality time step is done by computing locally the minimum travel time for each network portion, then obtaining the minimum of these values (this involves again an `MPI_Allreduce` operation).

4. Results

The parallel algorithm for the water quality simulation has been tested over a platform formed by several Pentium PRO 200 MHz PCs with Windows o.s.

connected via a Fast Ethernet network. Two water networks, named *Test A* and *Test B*, have been used for the testing. Their main characteristics can be seen in Table 1.

Table 1. Characteristics of the test networks.

	Pipes	*Nodes*	*Tanks*	*Substance*	*Simulation duration*
Test A	4901	2501	1	Chlorine	48.00 hrs
Test B	19801	10001	1	Chlorine	24.00 hrs

Execution times obtained with these test networks are shown in Fig. 2, which also includes the execution times of the original sequential EPANET 1.1e simulation program.

The resulting speed-up is shown in Fig. 3. Here, the speed-up values are taken with respect to the sequential simulation program EPANET, in order to get the real gain in execution time that has been achieved. A speed-up of up to 3.1 has been obtained, which illustrates the good performance achieved with the parallelization.
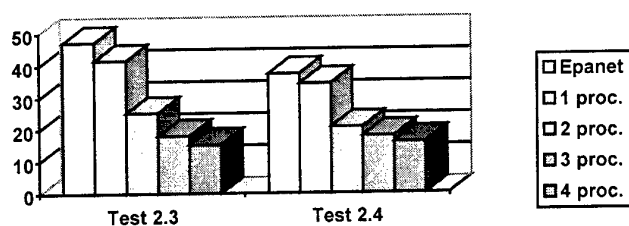
Fig. 2. Parallel algorithm execution times, in seconds.
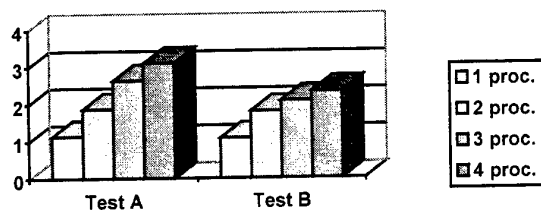
Fig. 3. Parallel algorithm speed-up.

Finally, efficiency obtained can be seen in Table 2. In this case the efficiency is obtained with respect to the parallel algorithm executed on a single processor. It can be seen that Test B presents lower efficiencies than Test A, although Test B corresponds to a larger network. This is due to the time spent on reading and distributing hydraulic results, and collecting and writing the final results, which is a process that must be done at the beginning and the end of each hydraulic time step.

Table 2. Parallel algorithm efficiency.

Eficiency	2 proc.	3 proc.	4 proc.
Test A	0,82	0,77	0,68
Test B	0,83	0,64	0,54

5. Conclusions

A parallel algorithm for the quality simulation of drinking water networks, based on the DVEM method implemented in the EPANET package, has been presented. The proposed method allows for efficient simulation of the spatial and temporal distribution of substances in water networks.

The algorithm has been developed in the frame of the HIPERWATER project. The objective of HIPERWATER has been to meet the need of computational power by introducing High Performance Computing techniques. The project considered the problems of hydraulic simulation and leakage minimization, as well as the water-quality simulation.

Concerning the water-quality algorithm presented here, results obtained show an important reduction in the computation time with respect to the EPANET package. The paper shows that *High Performance Computing* is a valuable tool for the reduction of time spent on quality simulations for large drinking water networks.

References

1. BUI, T., and JONES, C. (1993). "A Heuristic for Reducing Fill in Sparse Matrix Factorisation", *6th SIAM Conf. Parallel Processing for Scientific Computing*, 711-718.
2. HENDRICKSON B., and LELAND R. (1993). "A Multilevel Algorithm for Partitioning Graphs". *Technical Report SAND93-1301*, Sandia National Laboratories.
3. HERNÁNDEZ, V., MARTÍNEZ, F., VIDAL, A.M., ALONSO, J.M., ALVARRUIZ, F., GUERRERO, D., RUIZ, P.A., VERCHER, J. (1999a). "HIPERWATER: A High Performance Computing EPANET-Based Demonstrator for Water Network Simulation and Leakage Minimisation", International Conference on Computing and Control for the Water Industry (CCWI'99), Exeter, UK.
4. HERNÁNDEZ, V., VIDAL, A.M., ALVARRUIZ, F., ALONSO, J.M., GUERRERO, D., RUIZ. P.A. (1999b). "HIPERWATER: A High Performance Computing Demonstrator for Water Network Analysis", Parallel Computing'99 (ParCo'99), Delft, The Netherlands.
5. ROSSMAN, L. A. (1993a). *EPANET User's Manual*, US Environmental Protection Agency.
6. ROSSMAN, L. A., BOULOS, P. F., and ALTMAN, T. (1993b). "Discrete Volume-Element Method for Network Water-Quality Models", *J. of Water Resources Planning and Management*, ASCE, 119(5), 505-517.

A visualization tool for the performance prediction of iterative methods in HPF

F.F. Rivera, J.J. Pombo, T.F. Pena, D.B. Heras,
P. González, J.C. Cabaleiro, and V. Blanco

University of Santiago de Compostela
Dept. Electronics and Computing,
15706, Spain
`fran@dec.usc.es`

Abstract. An exhaustive library of sparse iterative methods and pre-conditioners in HPF was developed, and a tool to predict and visualize the performance of these codes is presented. This tool can be used both by the users and by the library's developers to optimise the efficiency of the codes, as well as to simplify their use. The information offered by this tool combines theoretical features of the methods and preconditioners in addition to some practical considerations and predictions about performance aspects of their execution.

1 Introduction

The complexity of parallel systems makes a priori performance prediction difficult. In fact, performance instrumentation and visualization in parallel systems was found to be a complex multidimensional problem [9]. A performance data collection, analysis and visualization environment is needed to detect the effects of architectural and system software variations.

The reasons for poor performance of codes on distributed memory systems can be varied, and users need to be able to understand and correct performance problems. This fact is specially relevant when high level libraries and programming languages are used to implement parallel codes, as in the case of HPF [7].

Most of the performance tools, both research and commercial, focus on low level message-passing platforms like MPI or PVM [4] [5] [1], and the most prevalent approach taken by these tools is to collect performance data during program execution and then provide post-mortem display and analysis of performance information [10] [11]. Our proposal is different, we present a tool that predicts performance data of irregular HPF codes before executing them.

The efficient implementation of irregular codes in HPF is hard. However, several techniques for handling this problem using intrinsic and library procedures as well as data distribution directives can be applied. An exhaustive library of iterative methods and preconditioners was developed [3], the tool presented in this paper analyses the performance of these codes. This tool can be used both by

2

the users of this library to optimize the efficiency and by the library's developers to check the inefficiencies in an easy to use interface.

Several strategies were used to optimize the efficiency of these parallel codes. In the literature, many iterative methods have been presented and developed and it is impossible to cover them all. We chose the methods below, either because they represent the current state of the art for solving large sparse linear systems [2] or because they present special programming features. The methods we consider are: Conjugate Gradient (CG), Biconjugate Gradient (BiCG), Biconjugate Gradient Stabilized (BiCGSTAB), Conjugate Gradient Squared (CGS), Generalized Minimal Residual (GMRES), Jacobi, Quasi-Minimal Residual (QMR) and Gauss-Seidel Successive Over-Relaxation (SOR). Additionally, some preconditioners were also implemented in HPF, and can be applied to the target sparse matrix to transform it into one with a more favourable spectrum. These preconditioners are: the Jacobi preconditioner, the Symmetric Successive Over-Relaxation (SSOR), the Incomplete LU factorization (ILU(0)), the Incomplete LU factorization with threshold (ILUT), the Neumann Polynomial preconditioner and the Least Squares Polynomial preconditioner.

The system on which we implemented the parallel codes was the Fujitsu AP3000, a distributed memory multiprocessor which consists of 12 UltraSparc processors connected by a mesh network [8]. However, both, the parallel codes and the performance tool, can be directly used on other parallel and distributed platforms with minor changes if any.

2 The visualization tool

Some knowledge about the linear system is needed to guarantee convergence of these algorithms, and generally the more that is known the more the algorithm can be tuned. Thus, we have chosen to present an algorithmic outline, with guidelines for choosing a method as part of our tool.

A method that works efficiently for one problem may not work so good for another. This problem increases in complexity if the application of some preconditioner is also considered. The tool presented in this report helps to find the most effective method for the matrix in hand avoiding the need of an exhaustive searching.

Our proposal combines theoretical features of the methods and preconditioners in addition to some practical considerations. In this way, relationships between data become readily apparent when the data are graphically displayed. The tool aids users in understanding, and drawing conclusions from the iterative methods and their implementation in HPF for each particular matrix.

The goals obtained by our prototype are:

- The tool allows users to select interactively the data to be displayed.
- The tool is easy to install and its use is fairly self-explanatory.
- It includes tools for gathering performance information.
- The individual analysis and visualization components are easy to build for many different matrices and preconditioners.

- As a standard platform, TCL/TK, was used to implement the tool, new components or modifications can be added.
- The tool is fast because the great amount of data required is filtered.
- It provides great functionality as it uses a Xwindows platform.
- It supports multiple analysis levels, including the sparse matrix characteristics, the methods, and the performance analysis and predictions.
- It is portable to systems including a TCL/TK library, providing portability across a great variety of computers.
- Although the target platform is the Fujitsu AP3000, the tool can be easily adapted to analyse other multiprocessors.

The number of generated events is potentially enormous. The environment includes a set of data filters that process the input data reducing their number. Via an environment control, the display can be changed dynamically, allowing the user to select the best suited display formats to the data.

The diversity of the performance data demands an equally rich set of performance displays. The displays include: dials, bar charts, LEDs, Kiviat diagrams, matrix views, X-Y plots, 3-dimensional plots and text information.

The user interface for the prototype visualization system provides comprehensive control. Through menus the user can obtain valuable information about the execution of the iterative method and preconditioner to select the best one in a friendly environment.

The main capabilities of this visualization tool are:

- It loads the sparse matrix in a standard format [6] and determines its essential characteristics, like, the pattern, the sparsity, the bandwidth, the symmetry, etc.
- Theoretical aspects about the application of the iterative methods and preconditioners to the matrix.
- The number of floating point operations required for each iteration of the methods and preconditioners for both, the sequential and the HPF codes.
- The load balance in terms of the computational costs.
- The number of communications and their lengths. This information can be shown for each processor.
- A prediction about the execution time for each iteration.
- As the number of iterations required by any method can not be predicted, a small number of iterations could be executed in order to analyse changes in the residuals, and get a first approach about the convergence of each method and preconditioner.
- Detailed statistical information about a routine can be seen.
- The use of pull-down menus to select visualization displays, or to change options is available.
- The statistics display shows the cumulative data for the complete parallel code or for each process.
- Finally, the method and the preconditioner can be actually executed.

4

The snapshots in Figure 1 show an example of the tool, in particular it shows the main menus, the menu for selecting a sparse matrix from a file, the pattern of this matrix, the help window and the window for selecting the number of processors to execute the parallel code.

And, in Figure 2, note the performance consultant window that shows the statistics for each process, a Kiviat graph showing the load balance, the histogram of the number and length of the messages to be sent and received by each processor, and the window for selecting the iterative method and preconditioner.
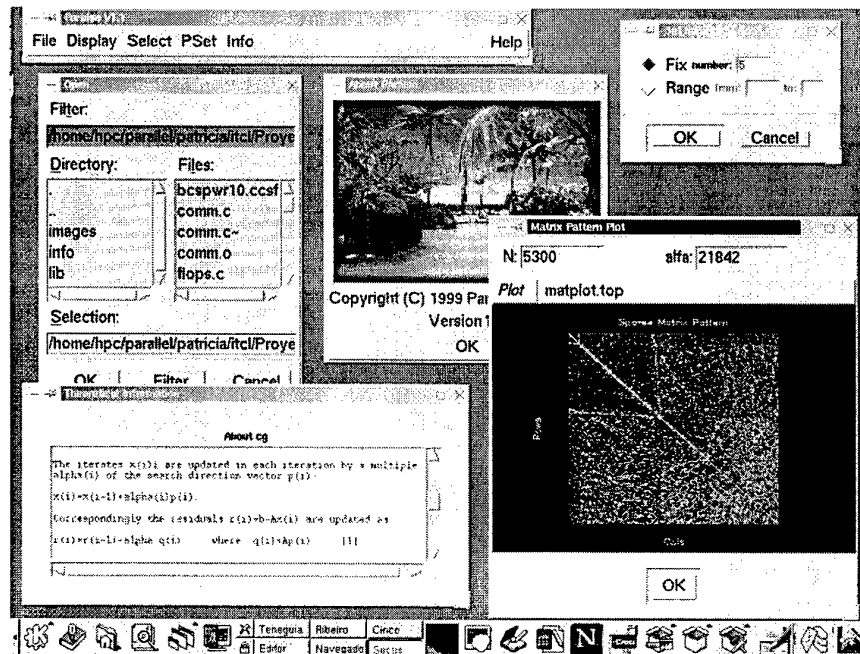
Fig. 1. Example of use of the visualization tool.

References

1. Allan, R. J., Heggarty, J., Goodman, M. and Ward, R. R.: Parallel Application
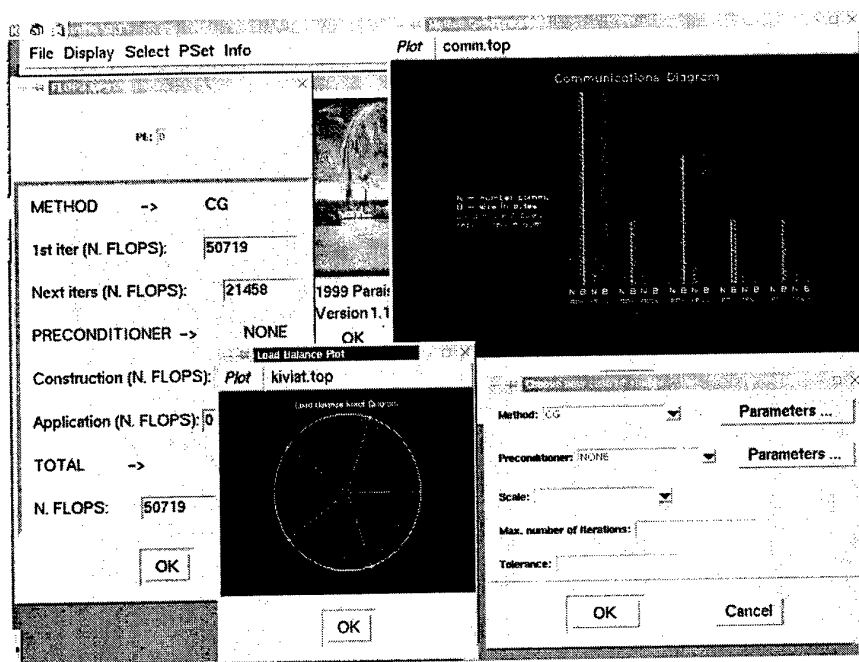
Fig. 2. Example of some results shown by the visualization tool.

Software on High Performance Computers. Survey of Parallel Performance Tools and Debuggers. http://www.cse.clrc.ac.uk/Activity/HPCI

2. Barret, R., Berry, M., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., Vorst, H.: Templates for the solution of linear systems: building blocks for iterative methods. SIAM (1994)

3. Blanco, V., Cabaleiro, J. C., González, P., Heras. D. B., Pena, T. F., Pombo, J. J. and Rivera, F. F.: A performance analysis tool for irregular codes in HPF. Fifth European SGI/Cray MPP Workshop, Bologna (1999).

4. Browne, S., Dongarra, J., London, K.: Review of performance analysis tools for MPI parallel programs http://www.cs.utk.edu/ browne/perftools-review/

5. dimemas: http://www.cepba.upc.es/ luisg/paraver/dip/dimemas.html

6. Duff, I.S., Grimes, R.G., Lewis, J.G.: Users guide for the harwell-boeing sparse matrix collection. Techn. Report TR-PA-92-96, CERFACS (1992)

7. High Performance Fortran Forum: High Performance Fortran language specification. (1997)

8. Ishihata, H., Takahashi, M., Sato, H.: Hardware of AP3000 Scalar Parallel Server. Fujitsu Sci. Tech. (1997) 24-30

9. Simmons, M., Koskela, R.: Performance instrumentation and visualization. ACM Press. (1990)

10. vampir: http://www.pallas.com

6

11. Yan, J., Sarukhai, S., Mehra, P.: Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit. Software-Practice and experience 25:4. pp. 429-461. (1995)

A Methodology for Designing Algorithms to Solve Linear Matrix Equations

Gloria Martínez[1], Germán Fabregat[1], and Vicente Hernández[2]

[1] Dpt. de Informática, Univ. Jaume I, Campus Riu Sec
E-12071 Castellón, Spain
{martine, fabregat}@inf.uji.es
[2] D.S.I.C., Univ. Politécnica de Valencia, Camino de Vera s/n
E-46071 Valencia, Spain
vhernand@dsic.upv.es

Abstract. We present a systematic and simple methodology to design parallel algorithms to solve the Generalized Sylvester Equation and other linear matrix equations. The resulting algorithms are well suited to be implemented using standard libraries of matrix arithmetic routines.

1 Introduction.

The solution of the Generalized Sylvester Equation, $AXB + CXD = E$, with $A, C \in R^{m \times m}$, $B, D \in R^{n \times n}$ and $X, E \in R^{m \times n}$, has wide application in modern Linear Control Theory [7],[8],[13]. When addressing particular problems, simpler equations derived from it, as the Sylvester[8],[13],[2], Lyapunov [14] and Stein [8],[13] equations are also frecuently used.

In this paper we introduce a systematic and simple design methodology to solve them, deriving algorithms directly expressed in terms of basic operations of Linear Algebra [4], so they can be easily implemented using standard scientific libraries. The methodology is based on the definition of the Kronecker Product, presented in section 2, and on the Back Substitution Algorithm. The parallelization of this algorithm and the basic operations of Linear Algebra is widely studied [4],[1],[15], so the methodology allows to systematically obtain parallel implementations of the resulting algorithms. The proposed methodology has been already tested in the design of a library of systolic routines [10], using dynamic arrays and applying the DBT transformation [12] on the basic operations.

To simplify the description of the methodology with a practical example in section 3, we will assume a triangular or quasi triangular form of the equation. Results for the general case are presented in [9],[11].

2 The Kronecker Product and the Vector Function.

Given the $A = [a_{ij}] \in R^{m \times m}$ and $B = [b_{ij}] \in R^{n \times n}$ matrices, the **Right Kronecker Product of A and B**, written $A \otimes B$, is defined as the block

matrix,

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B & \cdots & a_{1m}B \\ a_{21}B & a_{22}B & \cdots & a_{2m}B \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mm}B \end{pmatrix} = [a_{ij}B] \in R^{mn \times mn}. \tag{1}$$

Given $A \in R^{m \times m}$, $A = (A_{:,1}, A_{:,2}, \ldots, A_{:,n})$ where $A_{:,i} \in R^m$, with i = 1, 2,..., n, the vector

$$\begin{pmatrix} A_{:,1} \\ A_{:,2} \\ \cdots \\ A_{:,m} \end{pmatrix} \in R^{mn \times 1}, \tag{2}$$

is called **Vec-function of A** and written $vec(A)$. Among the properties of the Vec-function, the following two [5]

1. $\forall\ A, B \in R^{m \times n}$ and $\forall\ \alpha, \beta \in R$, $vec(\alpha A + \beta B) = \alpha vec(A) + \beta vec(B)$,
2. If $A \in R^{m \times m}$, $B \in R^{n \times n}$ and $X \in R^{m \times n}$, then $vec(AXB) = (B^T \otimes A)vec(X)$,

will allow to use the Kronecker Product as a tool to solve the studied matrix equations and design the corresponding algorithms.

3 Application of the Kronecker Product and the Vec-Function.

Previous to the application of the methodology, the problem is transformed into a condensed form according to the method proposed by Golub, Nash and Van Loan [3]. Applying the previous definitions to the *Triangular* case of equation $AXB + CXD = E$, we obtain the linear equation system $(B^T \otimes A + D^T \otimes C)vec(X) = vec(E)$, shown in figure 1. Its block structure suggests the use of the Back Substitution Algorithm to solve the problem [6] but this implementation has always been discarded due to the huge size of the resulting system. Our methodology uses the Kronecker Product in the design phase to study the structure of the resulting system, having figure 1 as the starting point for the design of the algorithms. Adapting the Back Substitution to the corresponding block structure, we obtain the **SGT Algorithm** shown in figure 2, that uses basic operations of the Linear Algebra: Solve a system, Gaxpy and Saxpy.

For the *Quasi-Triangular* case, we assume that the pencil $A - \lambda C$ is reduced to the Real Schur Form, and the pencil $D - \lambda B$ to the Triangular Form: each block in figure 1 is a Schur matrix. The main difference is now that the matrix $(Ab_{ii} + Cd_{ii})$ must be triangularized before solving the value of x_i. So adding to the SGT Algorithm the new operation *Calculate Q* : $(Ab_{ii} + Cd_{ii})Q$ *is upper triangular*, whose outputs are the matrices AQ, CQ (with the same zero-structure that matrix A) and Q (tridiagonal), we obtain the **SGH Algorithm**, shown in figure 3. The election of a column-oriented transformation has been made to optimize the data flow for a systolic implementation.
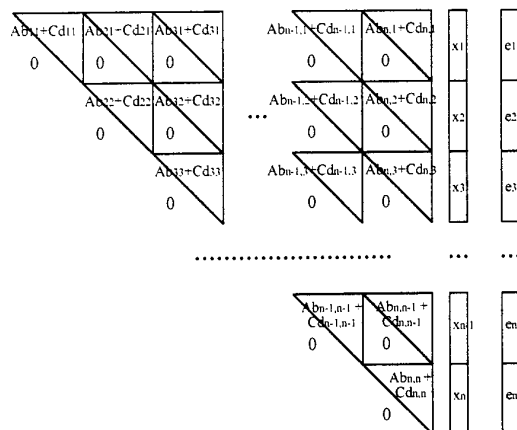
Fig. 1. Linear Equation System obtained by applying the Kronecker Product and the Vec-function to the Triangular Generalized Sylvester Equation.
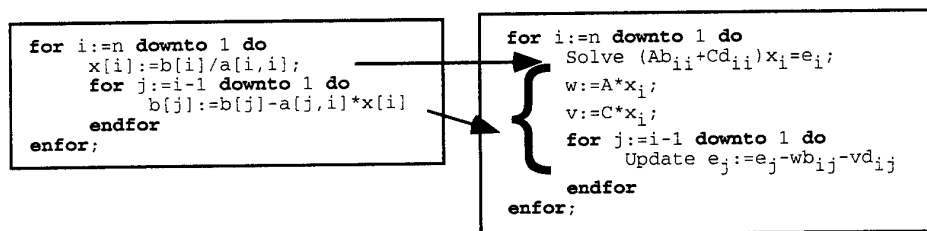
Fig. 2. Transformation of the Back Substitution Algorithm into the SGT Algorithm.
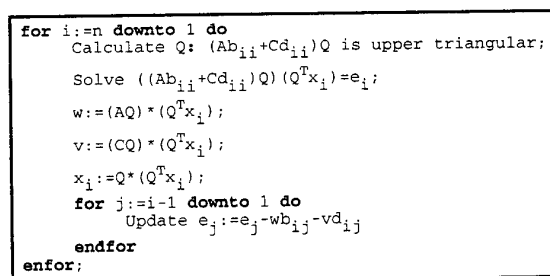
Fig. 3. SGH Algorithm.

3.1 Obtaining Block Algorithms.

Starting from figure 1 it is also possible to design algorithms for $N \times N$ upper triangular blocks of size $M \times M$ each, being $N = pn$ and $M = qm$. Each of these generic blocks, $Ab_{ij} + Cd_{ij}$, with j=1..N and i=j..N, presents the structure shown in figure 4: they are built of $q \times q$ subblocks of size $m \times m$. Therefore each of the columns of X and E is also built of q blocks of size m. We will denote the subblock at the r row and s column from the $(Ab_{ij} + Cd_{ij})$ block as $(A^{rs}b_{ij} + C^{rs}b_{ij})$; and the rth subvector from the ith column of X, x_i, or E, e_i, as x_i^r or e_i^r, respectively. From the described block structure, it is possible to
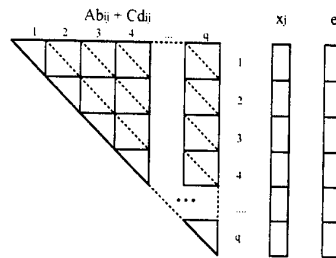
Fig. 4. Structure of each triangular block from the coefficient matrix.

obtain different algorithmic schemes. After studying several possibilities for the *Triangular* case, we have chosen to rewrite block-oriented versions of the **Solve**, **Gaxpy** and **Update** operations. The two obtained algorithms, shown in figure 5, are called **SGTB2.1** (column-oriented) and **SGTB2.2** (row-oriented).

```
for i:=N downto 1 do
    for s:=q downto 1 do
        w^s:=0; v^s:=0
    endfor;
    for s:=q downto 1 do
        e_i^s:=e_i^s-w^s b_ii-v^s d_ii ;
        Solve (A^ss b_ii+C^ss d_ii)x_i^s=e_i^s;
        for r:=s downto 1 do
            w^r:=w^r+A^rs*x_i^s;
            v^r:=v^r+C^rs*x_i^s
        endfor;
        for j:=i-1 downto 1 do
            Update e_j^s:=e_j^s-w^s b_ij-v^s d_ij
        endfor
    endfor
endfor;                             (a)
```

```
for i:=N downto 1 do
    for s:=q downto 1 do
        w^s:=0; v^s:=0;
        for r:=q downto s+1 do
            w^s:=w^s+A^sr*x_i^r;
            v^s:=v^s+C^sr*x_i^r
        endfor;
        e_i^s:=e_i^s-w^s b_ii-v^s d_ii ;
        Solve (A^ss b_ii+C^ss d_ii)x_i^s=e_i^s;
        w^s:=w^s+A^ss*x_i^s;
        v^s:=v^s+C^ss*x_i^s;
        for j:=i-1 downto 1 do
            Update e_j^s:=e_j^s-w^s b_ij-v^s d_ij
        endfor
    endfor
endfor;                             (b)
```

Fig. 5. (a) SGTB2.1 Algorithm, (b) SGTB2.2 Algorithm.

The main difference between the *Triangular* and the *Quasi-Triangular* cases resides on the division of each Schur block to obtain a block-oriented version of the operation **Calculate Q**. We must apply the division depicted in figure 6: two consecutive blocks in a row, $(A^{rs}b_{ii} + C^{rs}d_{ii})$ and $(A^{r,s+1}b_{ii} + C^{r,s+1}d_{ii})$, share a column, to correctly nullify the subdiagonal elements. The resulting **SGHB1.1** and **SGHB1.2** algorithms are shown in figure 7. The blocks affected for this special division are marked with bold type. Note that although the real size of each submatrix $((A^{ss}b_{ii} + C^{ss}d_{ii})Q^s)$, in the operation **Solve** is $m \times (m+1)$, the result $(Q^s)^T x_i^s$ must be of size m. Therefore some updates are deferred until the corresponding element is calculated.
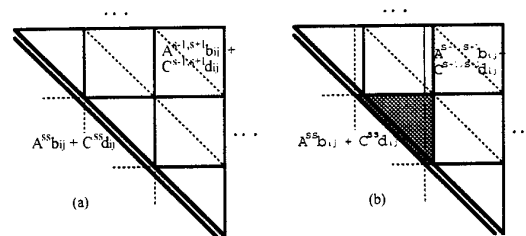
Fig. 6. (a) Block division for Solve and Gaxpy operations. (b) Block division for Calculate Q and Apply Q operations.

4   Conclusions

We have presented a methodology that allows the simplification of a complex problem to be solved using basic Linear Algebra operations and implementing the solution using standard libraries, and that can also be used to obtain block algorithms. The methodology itself is a powerful graphical tool that helps the design by offering a clear representation of the data flow and dependencies. Therefore the data flow is adapted to the processing requirements, eliminating the need for intermediate storage resources. The methodology has been applied to other equations [9] obtaining a reduced set of basic arrays that form a complete Systolic Library [10] for solving a wide variety of problems in the field of matrix algebra (see, e.g., [11]).

References

1. Bertseklas, P.H., Tsisiklis, M.M.: Parallel and Distributed Computations. Prentice-Hall Int. Eds. (1989)
2. Cavin, R.K., Bhattacharyya, S.P.: Robust and Well-Condicioned Eigenstructure Assignment via Sylvester's Equation. Opt. C. Appls. & Meths., Vol.4 (1983) 205–

```
for i:=N downto 1 do
  for s:=q downto 1 do
    w^s:=0; v^s:=0
  endfor;
  for s:=q downto 1 do
    e_i^s:=e_i^s-w^s b_ii-v^s d_ii ;
    Calculate Q^s:(A^ss b_ii+C^ss d_ii)Q^s is upper triang;
    Solve ((A^ss b_ii+C^ss d_ii)Q^s)(Q^s)^T x_i^s=e_i^s;
    w^s:=w^s+(A^ss Q^s)*(Q^s)^T x_i^s;
    v^s:=v^s+(C^ss Q^s)*(Q^s)^T x_i^s;
    if (s>1) then
      for r:=s-1 downto 1 do
        Apply Q^s to matrices A^rs and C^rs;
        w^r:=w^r+A^rs Q^s*(Q^s)^T x_i^s;
        v^r:=v^r+C^rs Q^s*(Q^s)^T x_i^s
      endfor
    endif;
    if (s<q) then Obtain w1 and v1 from [x_i^{s+1}]_0 endif;
    for j:=i-1 downto 1 do
      Update e_j^s:=e_j^s-w^s b_ij-v^s d_ij
      if (s<q) then
        Update [e_j^{s+1}]_1:=[e_j^{s+1}]_1-w1 b_ij-v1 d_ij
      endif
    endfor
  endfor                                        (a)
endfor;
```

```
for i:=N downto 1 do
  for s:=q downto 1 do
    w^s:=0; v^s:=0
    if (s<q) then
      for r:=q downto s+1 do
        Apply Q^r to matrices A^sr and C^sr;
        w^s:=w^s+(A^sr Q^r)*((Q^r)^T x_i^r);
        v^s:=v^s+(C^sr Q^r)*((Q^r)^T x_i^r)
      endfor
    endif;
    e_i^s:=e_i^s-w^s b_ii-v^s d_ii;
    Calculate Q^s:(A^ss b_ii+C^ss d_ii)Q^s is upper triang;
    Solve ((A^ss b_ii+C^ss d_ii)Q^s)((Q^s)^T x_i^s)=e_i^s;
    w^s:=w^s+(A^ss Q^s)*((Q^s)^T x_i^s);
    v^s:=v^s+(C^ss Q^s)*((Q^s)^T x_i^s);
    if (s<q) then Obtain w1 and v1 from [x_i^{s+1}]_0 endif;
    for j:=i-1 downto 1 do
      Update e_j^s:=e_j^s-w^s b_ij-v^s d_ij;
      if (s<q) then
        Update [e_j^{s+1}]_1:=[e_j^{s+1}]_1-w1 b_ij-v1 d_ij
      endif
    endfor
  endfor                                        (b)
endfor;
```

Fig. 7. (a) SGHB1.1 Algorithm. (b) SGHB1.2 Algorithm.

3. Golub, G.H., Nash, S. Van Loan, C.F.: A Hessenberg-Schur Method for the Problem $AX + XB = C$. IEEE Trans. on Aut. Control, Vol. AC-24, **6** (1979) 909–

4. Golub, G.H., Van Loan, C.F.: Matrix Computations, 2nd edition. John Hopkins University Press, Baltimore (1989)

5. Lancaster, P.: The Theory of Matrices. Academic-Press, New York (1969)

6. Lancaster, P.: Explicit Solutions of Linear Matrix Equations. SIAM Review, Vol. 12, **4** (1970) 544–

7. Luenberger, D.G.: Time-Invariant descriptor Systems. Automatica, **14** (1978) 473–

8. Laub, A.J.: Numerical Linear Algebra Aspects of Control design Computations. IEEE Trans. on Aut. Control, Vol. AC-30, **2** (1985) 97–

9. Martínez, G.: *Algoritmos Sistólicos para la Resolución de Ecuaciones Matriciales Lineales en Sistemas de Control*. Ph.D. Thesis, U. P. de Valencia, Spain (1999)

10. Martínez, G., Fabregat, G. Hernández, V.: *Une Librairie de Routines Systoliques*. Proceedings of Renpar '11 (1999) 181–186.

11. Martínez, G., Fabregat, G. Hernández, V.: Solving the Generalized Sylvester Equation with a Systolic Library. To appear in Proceedings of VecPar'2000.

12. Navarro, J.J., Llabería, J.M., Valero, M.: Partitioning: an Essential Step in Mapping Algorithms into Systolic Array Processors. IEEE Computer, July (1987) 77–

13. Petkov, P.H., Christov, N.D., Konstantinov, M.M.: Computational Methods for Linear Control Systems. Prentice-Hall, Hertfordshire (1991)

14. Sima, V.: Algorithms for Linear-Quadratic Optimization. Marcel Dekker Inc., New York (1996)

15. Van de Geijn, R.: Using PLAPACK: Parallel Linear Algebra Package. MITPRESS (1997), *http://www.cs.utexas.edu/users/plapack*

A new user-level threads library: *dthreads*

A. García Dopico, A. Pérez Ambite, and M. Martínez Santamarta

Dept. Arquitectura y Tecnología de Sistemas Informáticos,
Facultad de Informática, Universidad Politécnica de Madrid,
Boadilla del Monte, 28660, Madrid, Spain
Tel: 34 91 336 6603, Fax: 34 91 336 7376
{dopico, aperez, mmartinez}@fi.upm.es
http://www.datsi.fi.upm.es

Abstract This paper describes the design and implementation of *dthread*, a new general purpose user-level threads package, designed to support fine-grain parallel applications in a portable and efficient way. We decided to build this new library because the performance of the Solaris threads library is not good enough to support fine-grain parallel applications. We include some measurements comparing the performance of both libraries. They show our objective has been reached.

Topics: Parallel and distributed algorithms, Operating systems.

Keywords: Threads, Parallelism, Solaris, Multiprocessors.

1   Introduction

This work is part of a large project: VAMOS, "VHDL Advanced Multiprocessor Optimized Simulation", developed by the Computer Architecture Department (UPM) and the TGI company. The objective of VAMOS was to develop a VHDL parallel simulator for shared memory multiprocessors. This parallel simulator runs on Solaris multiprocessors and uses fine and very fine-grain parallelism.

Due to the poor performance we observed in the Solaris threads library with this kind of parallelism, we decided to develop our own threads library to improve the performance. We have got a small, efficient, portable and standard threads library suitable for fine-grain parallelism.

2   State of the art

Initially threads were lightweight processes executing in a single address space that could run independently and concurrently. They were managed in the operating system kernel (kernel threads), which made threads expensive.

Later on user-level threads were introduced ([1, 2]). They have performance and flexibility advantages over kernel threads because they are managed within the user address space. But they have also disadvantages when a user-level thread performs blocking system calls or in presence of multiprogramming. These problems arise because there are two places where the next running thread can be

2   A. García, A. Pérez, M. Martínez

scheduled, one in the application and another one in the operating system, with very little coordination among them. This is called the two-level scheduling problem ([3–7]).

At the moment, the threads are supported by most of the operating systems, including Solaris, they are well-known and there is a standard ([8]) that assures their portability.

3   Solaris threads library

Solaris has two kind of threads: kernel-supported threads so called *Light Weight Processes* (LWP) and user-level threads, simply called *threads*. User-level threads are used to decrease the level of overheads involved in their management (creation, destruction, context switch,...). On the other hand, Solaris uses kernel threads as virtual processors to execute the user-level threads and to control the degree of real concurrency that the application requires.

Each LWP is independently dispatched by the kernel. They may run in parallel on a multiprocessor, being scheduled onto the available processors according to their scheduling class and priority. Threads are implemented by the library and are not known by the kernel.

We have found several problems in the Solaris user-level threads library that encouraged us to develop a new one:

- **Heavy weight**. User-level threads are quite heavy, being suitable for coarse-grain and middle-grain applications but never for fine-grain applications, because context switch involves heavy system calls that make context switch time usually longer than the tasks execution time.
- **Degree of concurrency**. The library changes dynamically and transparently the number of LWPs that give support to an application to solve the two level scheduling problem. When all the LWPs in the process are blocked in indefinite waits the kernel sends a signal to the threads library that responds creating a new LWP. Also the threads library makes LWPs to "ages" and, if they are not used for a long time, they are terminated. That means the user has a loose control over the actual degree of concurrency that is effective only in simple and small applications but has no control in a real, big application.
- **Poor locality of reference**. The library puts all the runnable threads together in a global queue. The LWPs always choose the first one, which implies a poor behavior in terms of locality of reference.
- **Bad optimizations**. Some library optimizations are very dependent on the application and quite often they decrease the performance instead of increasing it. For example, when a thread becomes blocked and there are no more runnable threads, the LWP that was running the thread must also stop running. It does so by waiting on an LWP semaphore associated with the thread (the LWP is parked), rather than idling on the global condition variable. This practice optimizes the case where the blocked thread becomes

runnable quickly, but leaves the application without one LWP for some time. Even it is possible to find some runnable threads waiting for a LWP while there are some parked LWPs.

4 *Dthreads* library

Once we identified the above mentioned problems, we concluded that the Solaris threads library was not appropriate for fine-grain parallelism. The best choice was to replace it with a new one, suitable for fine-grain parallelism. The principal goals were:

- **Efficiency**. This was the main objective. To accomplish it we reduced the threads weight to the minimum.
- **Threads management** is done exclusively in the user address space, without system calls.
- **Portability**. The *dthreads* library is POSIX compliant to ensure portability. Both *dthreads* and Solaris threads libraries can be used by the VHDL simulator by linking the chosen one.
- **Degree of concurrency**. The *dthreads* library leaves the control of the degree of concurrency that the application needs to the user and the modifications done are not transparent to the user.
- **Locality of reference**. This new library tries to avoid thread migration between LWPs, improving the efficiency of caches. To achieve this objective it has one local queue by processor and a global queue.
- **Avoid blocking system calls as much as possible**. If a user-level thread executes a blocking system call, the underlying kernel thread blocks too. Inside the application a virtual processor is lost and a physical one can be unused even if there are runnable user-level threads. In order to avoid processors being idle (two-level scheduling problem), the library implements a buffered input/output monitor. This monitor could also have been implemented with the Solaris threads library. The idea is to assure that most of the times the application threads will not block. The read and write operations are done by the monitor.

The architecture of the system based on *dthreads* is quite similar to the original one. The new user-level threads executes on the LWPs, that are used as virtual processors.

Each thread has a stack, an optional heap and a thread control block. The thread control block holds the thread identifier, a pointer to the stack and the thread context.

There is the possibility of assigning a heap to each thread to avoid contention in the dynamic memory allocation for high demanding memory applications. Threads can share memory dynamically allocated in any heap, but each thread must manage his own heap, allocating and freeing memory. If a thread doesn't know which one will free a memory block, it must use the global heap, that is lock protected to prevent concurrent access. There is no loss of generality

4   A. García, A. Pérez, M. Martínez

because local heaps are only an extension. The global heap is always present and his access is protected.

5 Results

Now we will show some measurements that justify the advantages of this new user-level threads library. The performance of this library is compared with the performance of the Solaris library to show the differences.

These measurements have been taken in a four processor `SunSPARCstation20`. It is based on 50 MHz SuperSparc processors with 128 Mbytes of shared memory.

Operation	Solaris	*dthread*	Speedup
Create/Destroy	2900 μseg	600 μseg	4.83
Lock/Unlock	1.7 μseg	0.68 μseg	2.5
GetSpecific	1.1 μseg	0.53 μseg	2.1
pthread_self	0.4 μseg	0.58 μseg	0.69

Table1. Performance.

Table 1 shows the time spent on several important operations both on *dthreads* and Solaris threads. However, these data are not enough to justify the development of a new user-level threads library. Tables 2 and 3 show the time used in a context switch with `pthread_yield` and with `conditions`. The differences between both libraries are very important.

Context switches between user-level threads can occur very often in a parallel application and can introduce important overheads in the Solaris threads library. With a single LWP, that is, without actual concurrency, Solaris threads management is done inside the user address space, without using system calls, and with reasonable times. However, in a regular situation, with several user-level threads over a few LWPs, Solaris introduces a lot of costly system calls with high overheads. The threads management is not done in the user address space anymore, and a lot of unnecessary system calls can appear (lwp_mutex_lock, lwp_mutex_wakeup, lwp_sema_post, lwp_sema_wait). The Dthreads library does not use system calls to manage user-level threads in any case, which explains the execution time differences.

There are some other reasons to build the *dthreads* library:

- To control the degree of concurrency. This library never changes the number of kernel threads that give support to the application unless the user asks for it.
- To improve the locality of references. Solaris uses only one global queue to put all the runnable threads. This solution gives an optimal load balance but a poor locality of references because a thread does not reuse its state present

Concurrency	Solaris	*dthread*	Speedup
1 Thread, 1 LWP	16.3 μseg	10.3 μseg	1.6
2 Thread, 1 LWP	24.7 μseg	10.3 μseg	2.4
4 Thread, 1 LWP	25.1 μseg	10.6 μseg	2.4
2 Thread, 2 LWP	7.2 μseg	6.0 μseg	1.2
4 Thread, 2 LWP	79.0 μseg	6.4 μseg	**12.3**

Table2. Context switch with *pthread_yield*.

Concurrency	Solaris	*dthread*	Speedup
1 Thread, 1 LWP	4.4 μseg	2.3 μseg	1.9
2 Thread, 1 LWP	38.4 μseg	13.3 μseg	2.9
4 Thread, 1 LWP	38.9 μseg	13.4 μseg	2.9
2 Thread, 2 LWP	103.8 μseg	18.0 μseg	5.8
4 Thread, 2 LWP	121.4 μseg	15.8 μseg	**7.7**

Table3. Context switch with *conditions*.

in the cache memory of the last processor where it ran. On the contrary, Dthread library has local queues to put each thread in the local runnable queue of the last processor where it ran.

– To increase the limit on the number of threads. Dthread library implements the threads with a small state that reduces the resources used and gives the ability to increase the number of threads that can be managed.

6   Conclusions

As has been shown, we have reached the starting objectives. We have got a general purpose user-level threads library for shared-memory multiprocessors that is POSIX compliant. It is efficient, small, portable, has good performance and it is suitable for fine-grain parallelism. It is faster than the Solaris user-level threads library and it solves the different problems that the Solaris threads library has.

References

1. B. N. Bershad; E. D. Lazowska; H. M. Levy, "Presto: A system for object-oriented parallel programming," *Software - Practice and Experience*, vol. 18, no. 8, Aug. 1988.
2. T.E. Anderson; E. D. Lazowska; H. M. Levy, "The performance implications of thread management alternatives for shared memory multiprocessor," *ACM Transactions on Computer Systems*, vol. 38, no. 12, Dec. 1989.
3. J. Edler; J. Lipkis; E. Schonberg, "Process management for highly parallel unix systems," in *Proceedings of the USENIX Workshop on Unix and Supercomputers*. September 1988, USENIX.

6   A. García, A. Pérez, M. Martínez

4. B. D. Marsh; M. L. Scott; T. J. LeBlanc; E. P. Markatos, "First-class user level threads," in *Proceedings of the 13th ACM Symposium on Operating Systems Principles.* October 1991, pp. 110–121, ACM.

5. T.E. Anderson; B. N. Bershad; E. D. Lazowska; H. M. Levy, "Scheduler activations: Effective kernel support for the user-level management of parallelism," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 53–79, Feb. 1992.

6. M. J. Feeley; J. S. Chase; E. D. Lazowska, "User-level threads and interprocess communication," Tech. Rep., University of Washington, 1993.

7. C. Koppe, "Sleeping threads: A kernel mechanism for support of efficient user level threads," Tech. Rep., University of Erlangen-Nürnberg, 1995.

8. IEEE Standard POSIX 1003.4a, *Threads Extension for Portable Operating Systems*, IEEE, Apr. 1993.

9. B. Cantarazo, *Multiprocessor System Architectures*, Prentice Hall, 1994.

10. M. L. Powell; S. R. Kleiman; S. Barton; D. Shah; D. Stein; M. Weeks, "Sunos multi-thread architecture," in *Proceedings of the Winter 1991 USENIX Conference*, Dallas, TX, 1991, USENIX.

11. IEEE, *Real-Time extensions to POSIX. IEEE Standard P1003.4*, Mar. 1993.

12. D. Keppel, "Register windows and user-space on the sparc," Tech. Rep., University of Washington, 1991.

13. D. Keppel, "Tools and techniques for building fast portable threads packages," Tech. Rep., University of Washington, 1993.

Grain Size Optimization of a Parallel Algorithm for Simulating a Laser Cavity on a Distributed Memory Multicomputer

Guillermo González-Talaván[1] and Luis A. M. Quintales[2]

[1,2] Departamento de Informática y Automática, Universidad de Salamanca
Plaza de la Merced s/n, 37008 Salamanca, Spain
Phone: +34-923294400{[1] ext. 1302, [2] ext. 1513}
{[1]gyermo, [2]lamq}@gugu.usal.es

Abstract. On this paper a parallel algorithm which allows an efficient calculation of a simulation of a laser cavity is presented. The optimal implementation of the algorithm on a distributed memory multicomputer results in the choice of an optimal grain size. This grain size must balance different factors depending on the parameters associated with the calculation. A model for the optimal choice of the grain size is proposed along with the corresponding experimental tests. The theoretical model can be easily extrapolated to a great number of similar problems.

Related topics: Parallel and distributed algorithms.

1 Purpose and scope of the work

The parallel algorithm that will be studied here relates to the simulation of the physical behaviour of a laser cavity. Laser (Light Amplifier by Stimulated Emission of Radiation) is a sort of light with certain optical properties such as high spatial and temporal coherence. An optical amplifier medium and two mirrors can produce the laser light. They make up a laser cavity as the one shown in fig. 1. See [1] for a practical application of this technology and [2] for a reference about the underlying physical problem.

The physical behaviour of the laser cavity can be simulated with a computer. In a semiclassical model, we need five functions to fully describe the state of the cavity on a given time t. Three of them are in connection with the matter: $p(x)$, $q(x)$, $w(x)$ and the other two, $\alpha(x)$ and $\partial\alpha/\partial t(x)$, with the radiation. The temporal evolution of the cavity state obeys to the partial differential equations (1).

$$\frac{\partial p}{\partial t} = -\omega_T\, q - \gamma_T\, p\ ,\quad \frac{\partial q}{\partial t} = \omega_T\, p + \alpha w - \gamma_T\, q\ ,$$

$$\frac{\partial w}{\partial t} = -\alpha q - \gamma_L\,(w-1)\ ,\quad \frac{\partial^2 \alpha}{\partial t^2} = c(t)^2\,\frac{\partial^2 \alpha}{\partial x^2} - G\,\frac{\partial^2 p}{\partial t^2}\ . \tag{1}$$
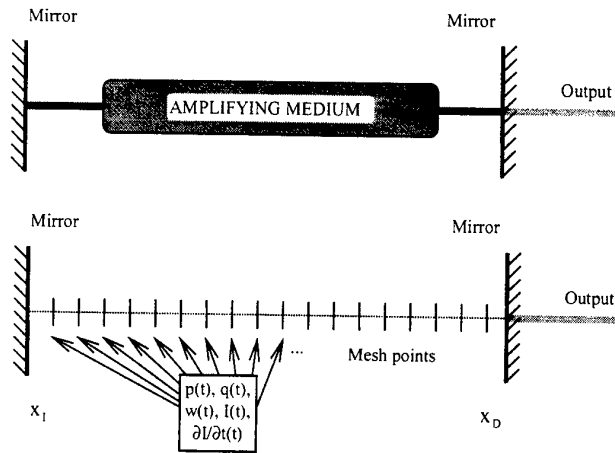
Fig. 1. Laser cavity diagram (up) and its discretization (down).

If a discretization of space and time dimensions is performed, one arrives to five arrays to describe the state of the cavity (fig. 1, down) and five equations of temporal evolution for them (2-6).

$$p(x,t+\Delta t) = p(x,t) + \frac{\partial p}{\partial t}(x,t)\cdot \Delta t + \frac{1}{2!}\frac{\partial^2 p}{\partial t^2}(x,t)\cdot (\Delta t)^2 + \frac{1}{3!}\frac{\partial^3 p}{\partial t^3}(x,t)\cdot (\Delta t)^3 \ldots \tag{2}$$

$$q(x,t+\Delta t) = q(x,t) + \frac{\partial q}{\partial t}(x,t)\cdot \Delta t + \frac{1}{2!}\frac{\partial^2 q}{\partial t^2}(x,t)\cdot (\Delta t)^2 + \frac{1}{3!}\frac{\partial^3 q}{\partial t^3}(x,t)\cdot (\Delta t)^3 \ldots \tag{3}$$

$$w(x,t+\Delta t) = w(x,t) + \frac{\partial w}{\partial t}(x,t)\cdot \Delta t + \frac{1}{2!}\frac{\partial^2 w}{\partial t^2}(x,t)\cdot (\Delta t)^2 + \frac{1}{3!}\frac{\partial^3 w}{\partial t^3}(x,t)\cdot (\Delta t)^3 \ldots \tag{4}$$

$$\alpha(x,t+\Delta t) = \alpha(x,t) + \frac{\partial \alpha}{\partial t}(x,t)\cdot \Delta t + \frac{1}{2!}\frac{\partial^2 \alpha}{\partial t^2}(x,t)\cdot (\Delta t)^2 + \frac{1}{3!}\frac{\partial^3 \alpha}{\partial t^3}(x,t)\cdot (\Delta t)^3 \ldots \tag{5}$$

$$\frac{\partial \alpha}{\partial t}(x, t + \Delta t) = \frac{\partial \alpha}{\partial t}(x, t) + \frac{\partial^2 \alpha}{\partial t^2}(x, t) \cdot \Delta t + \frac{1}{2!}\frac{\partial^3 \alpha}{\partial t^3}(x, t) \cdot (\Delta t)^2 \dots \tag{6}$$

Note that one must also take into account boundary conditions. The value of the arrays on either side of the cavity (i.e. where the mirrors lay) is zero for all t. With these relations, it is straightforward to device a sequential algorithm for simulating the cavity behaviour.
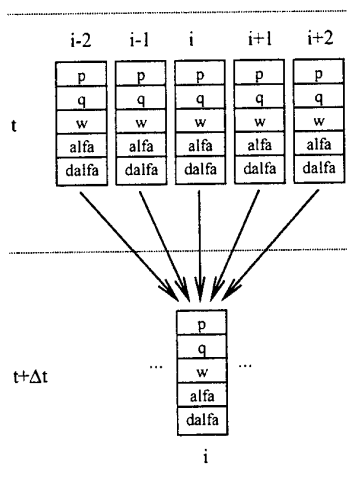
Fig. 2. Spatial dependencies for the calculations of the state of one point in the cavity in the next time step. Alfa and dalfa are the names of the arrays used in the program for α and $\partial \alpha / \partial t$.

When thinking about making a parallel algorithm to simulate the laser cavity physical behaviour, one must consider the spatial dependencies for calculating the value of the parameters in one point for the next time step in the future. Fig. 2 shows these dependencies for a certain point in the cavity.

A first approach to the parallel algorithm implementation can be the use of the "divide-and-conquer" techniques by dividing the cavity points in equal parts among the distinct processors that make up the multicomputer. Of course that, due to spatial dependencies, this partition must consider the boundary overlapping points necessary for calculations to be performed. The overlapping points should be at least two (eq. 2-6). The temporal evolution for problem solution forces a synchronization and the corresponding communication for interchanging boundary points between neighbouring processors (fig. 3, left).

Some initial studies demonstrated that depending on the cavity size and the number of processors used, this simple calculation plan could produce a low performance due to communication penalty introduced to the parallel algorithm. In other words, the grain size is too small for the work environment used.
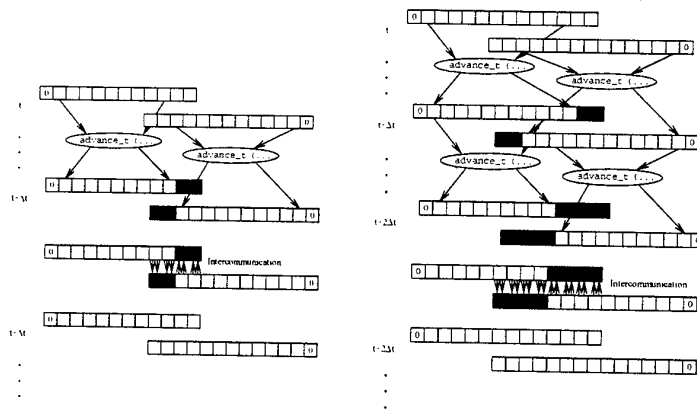
Fig. 3. Evolution diagram of the parallel algorithm with two processors (see text).

Thus, a first objective consists of designing a new algorithm that will allow the use of different sizes for the overlapping zone between processors so that the grain size can be increased to allow higher speedups. A working diagram of the new algorithm is presented in fig. 3, right.

A second objective would be thinking about some method for an adequate choice of the optimal grain size that can allow the best possible speedup as a function of the number of processors and the cavity size of the problem to be solved.

2 Fundamental Results Already Obtained

2.1 Hardware/Software Configuration

To do the parallel calculation, a PC network has been used. Each node has an Intel Pentium II Processor @ 266 MHz and 64Mb of RAM. Linux was used as the Operating System. Relating to communication, each computer has a Fast-Ethernet card connected to a switch. The message passing software used is MPI, in its LAM/MPI version 6.3.b2 implementation [3].
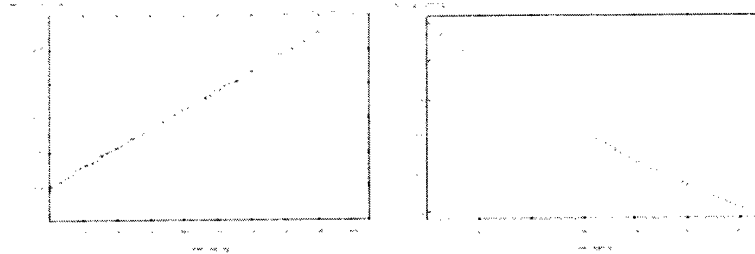
Fig. 4. Left: communication time vs. Overlapping for a 128000 point cavity and 16 processors. Right: maximum reachable speedup due to the algorithm overhead versus overlapping.

2.2 Grain Size Behaviour Issues

The new algorithm presented in 1.2 introduces different factors that will influence the final speedup in different ways. In fact, as the size of the overlapping zone increases:

1. The amount of information to be transmitted increases. An example of time spent in communication vs. size of the overlapping zone is presented in fig. 4, left.
2. The grain size grows, which is positive for the final speedup.
3. The parallel algorithm overhead increases, that is, some mesh points are calculated twice by different processors (fig. 3). This will yield a speedup decrease and a maximum reachable speedup as a function of the size of the overlapping zone. This speedup can be mathematically deduced and is plotted for an example in fig. 4, left.

As it was stated before, now we will propose a theoretical model that will allow us to predict the best overlapping size for a given cavity size and number of processors. Let us define a cycle as the time between successive intercommunications. We can define then the speedup as:

$$S = \frac{t_s}{t_p} = \frac{t_s}{t_p^{(calc)} + t_p^{(com)}} \tag{7}$$

where t_s stands for sequential time, t_p means parallel time, $t_p^{(calc)}$ is time spent in calculation for the parallel algorithm and $t_p^{(com)}$ is time spent in communication.

Considering a linear dependency of communication time on the overlapping size (as it suggests fig. 4, left), one can arrive to the following expression for the speedup:

$$S = \frac{\tau(NP-2)\frac{\sigma}{4}}{2(a+b\sigma)+\tau\left[\frac{NP-2}{N}-1+\frac{\sigma}{4}\right]\frac{\sigma}{4}} \tag{8}$$

where τ is the time spent on calculations of one point in the cavity, NP is the number of cavity points, σ is the overlapping size, $a+b\sigma$ is the linear dependence of communication time with σ, and N is the number of processors.

The overlapping size that maximizes (11) can be deduced from it and it is:

$$\sigma_{max} = \sqrt{\frac{32a}{\tau}} \tag{9}$$

2.3 Experimental results

Several experimental tests have been carried out in order to verify the theoretical model previously exposed. There is a good agreement between the calculated optimal overlapping and the experimental one in the cases we have analyzed. Some of these experimental results are shown in the following table:

# processors	Cavity size	$\sigma_{theoretical}$	$\sigma_{experimental}$
16	128000	37.2	34
16	40000	22.6	24

3 Conclusions

A parallel algorithm for a laser cavity simulation has been developed. This algorithm tries to obtain an optimal speedup by adequately selecting a grain size that balances the calculation/communication binomial. The optimal selection of the grain size is done by means of a very simple theoretical easy-to-calculate prediction which, additionally, could be extrapolated to similar algorithms for simulations of space/time evolution of physical systems in the future.

References:

1. Le Blanc, C.: Principes et realisation d'une source laser terawatt femtoseconde basée sur le saphire dopée au titane. Ph. Thesis. Ecole Polytechnique, Paris. (1995)
2. Allen, L., Eberly, J.H.: Optical Resonance and two-level atoms. Dover (1987).
3. Dongarra, J.J., Otto, S.W., Snir, M., Walker, D.: A Message Passing Standard for MPP and Workstations. Communications of the ACM, vol. 39, n° 7 (1997) 84-90

Running PVM Applications in the PUNCH Wide Area Network-Computing Environment

Dolors Royo[1], Nirav H. Kapadia[2], and José A. B. Fortes[2]

[1] Departemento de Arquitectura de Computedores,
Universitat Politècnica de Catalunya, Spain
dolors@ac.upc.es
[2] School of Electrical and Computer Engineering,
Purdue University, U.S.A.
{kapadia, fortes}@purdue.edu

Abstract. This paper outlines key issues that must be addressed in order to allow PVM-based programs to make effective use of resources within a wide area network-computing environment. Support mechanisms that allow unmodified PVM programs to be used within the PUNCH network-computing environment are also described. The mechanisms were found to be easy to implement, and preliminary experiences indicate that the described approach is well-suited for a network-computing environment.

1 Introduction

Distributed applications are often built on top of message-passing standards such as PVM [1] and MPI [2]. These standards were originally designed for relatively structured environments, where users are aware of all available machines and have direct access to them. In this context, the emerging wide area network-computing environment presents two interesting challenges: 1) the large size of the environment makes it difficult for users to keep track of all available resources, and 2) the dynamic and inter-institutional nature of the environment causes logistical problems when users are required to have actual user-accounts on all resources.

This paper describes how PVM- and MPI-based programs can make effective use of resources within a wide area network-computing environment by leveraging the functionality provided by PUNCH, the Purdue University Network Computing Hubs. A unique aspect of the described implementation is that neither the PVM/MPI programs nor the PVM/MPI libraries need to be modified.

PUNCH [3,4] is a distributed network-computing infrastructure that provides transparent and universal access to remote programs and resources via the World Wide Web. PUNCH users can define simulations, run them, and view the text and graphical output — all via their Web browsers. PUNCH currently provides access to more than fifty engineering software packages developed by thirteen universities and six vendors; a new program can be added in as little as thirty minutes. PUNCH can be accessed at www.ece.purdue.edu/punch.

The discussion in this paper focuses on PVM-based programs, but the ideas are equally applicable to MPI programs. The remaining sections are organized as follows. Section 2 outlines the issues that arise in the process of running a PVM program. Section 3 describes the support mechanisms provided by PUNCH for running PVM programs in a network-computing environment. Section 4 provides outlines related work. Finally, Section 5 presents concluding remarks and directions for future work.

2   Issues in Running PVM Programs

Running a PVM program in an environment where a user has direct access to all machines typically involves the following steps. The user must first select the machines for the given run and choose a "master" machine. Next, he/she must login to each "slave" machine and create a `.rhost` file that will allow PVM to start processes on that machine. After this, the user must create a PVM host file on the master machine; this file provides PVM with information about the available machines. Once this is done, the user must start the PVM daemons by invoking the PVM console on the master machine. At this point, the PVM system has been initialized and the user can start the PVM program.

In an environment where users are not aware of all available resources, the steps described above must be automated. In situations where users do not have user-accounts on all machines, operating system support for "scratch" accounts must be provided. The resulting sequence of steps required to start a PVM program in a wide area network-computing environment is illustrated in Figure 1.

3   The PUNCH Approach

PUNCH users initiate programs via a dynamically-generated Web interface that is accessible from standard WWW browsers [4]. For PVM- and MPI-based programs, users explicitly specify the number and types of machines required for a given run, in addition to other input parameters required by the program. This information is typically provided via menus and text-boxes in HTML forms.

When a user attempts to initiate a PVM- or MPI-based program, PUNCH first allocates the necessary resources using the user-supplied information about the number and type of machines. With reference to Figure 1, resource allocation (step 1 in the figure) involves two tasks: 1) selecting appropriate machines for the given run, and 2) ensuring that a scratch account is available for use on each of the selected machines.

The process of allocating resources is handled by PUNCH's pipelined resource management system, and proceeds as follows (see Figure 2). PUNCH first forwards the user-supplied information about machines to a local query manager. The query manager decomposes this information into individual components, each of which consists of a set of constraints (e.g., architecture, memory, need for scratch account, etc.) and a quantity (i.e., number of machines of this type). For example, a request for three Sun and four HP servers will be decomposed into two components, one for the three Suns and one for the four HPs.
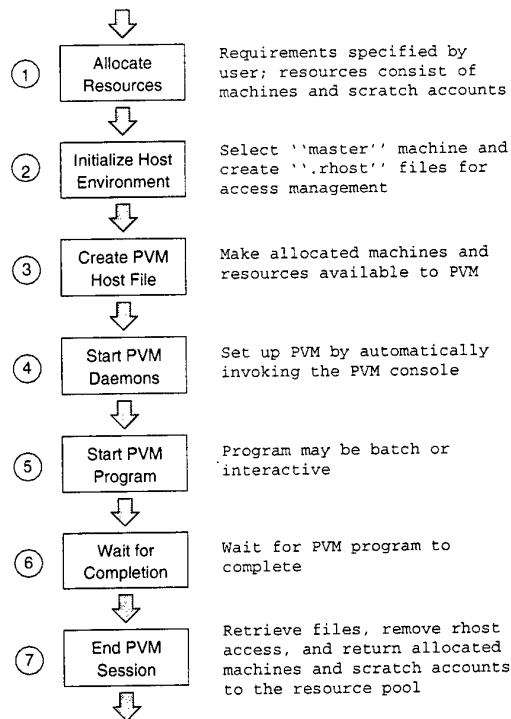
Fig. 1. The sequence of steps required to run a PVM program in a wide area network-computing environment. It is assumed that users are not aware of all available resources, and that they do not have user-accounts on all usable machines.

The individual components are then forwarded to the nearest (in terms of network reachability) pool manager(s), where they are processed concurrently. (If a pool manager is unable to satisfy a request, the query manager will forward the request to the next nearest pool manager.) The pool manager uses the constraints contained within a given query component to map it to an appropriate resource pool. A resource pool consists of 1) all machines in a specified local domain that satisfy a given set of constraints, and 2) scheduling agents that select machines from those within the pool on the basis of performance-related criteria (e.g., load balancing). For example, one pool could contain all Sun machines, another could contain all HP machines, a third could contain all Sun machines with at least 128MB memory, and so on. After a resource manager maps a query component to a resource pool, the scheduling agents associated with that pool allocate the desired number of machines and forward relevant information to another query manager stage (not shown in Figure 2; this stage is only required for queries that have to be decomposed into multiple components). The query manager reassembles the individual query components and forwards the results to PUNCH.
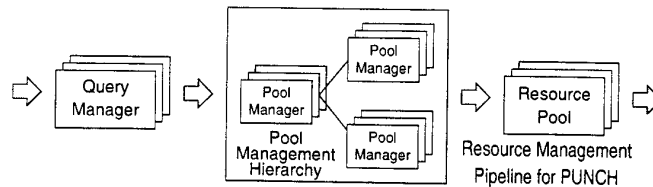
Fig. 2. The resource management pipeline utilized by PUNCH. A resource pool consists of dynamically aggregated resources that are similar in terms of a specified set of constraints and the associated scheduling agents.

A key feature of the resource management pipeline is that the resource pools are created dynamically from site-specific databases. Pool managers create resource pools when they are required and automatically destroy pools that have been inactive. This mechanism allows the resource management system to dynamically minimize the scheduling overhead for the specific types of jobs that are being initiated at any given time. Another benefit comes from the manner in which pool managers are chosen — because "closer" managers are selected first, allocated machines tend to be near each other. (In the current implementation, the closeness between managers is defined by a static quantity and machines controlled by the same manager are assumed to be at zero distance.)

Once the necessary resources have been allocated, the host environment is initialized as follows (step 2 in Figure 1). PUNCH first selects the "master" for this run — the first allocated machine is arbitrarily chosen for this role. Next, PUNCH uses secure shell (SSH [5]) to login to the allocated scratch account on each of the remaining allocated machines and creates the necessary .rhost files. If secure shell is not available on a given machine, rsh or rexec can be used instead. The key advantage of the PUNCH approach with respect to this step is that neither PVM nor the user need to be given access to the passwords for the scratch accounts. This allows PUNCH to recycle scratch accounts among users in a secure manner.

The third step involves generating a PVM host file that contains the names of the machines allocated for this run and the login names for the corresponding scratch accounts. PUNCH uses secure shell to access the scratch account on the "master" machine and writes the appropriate information into a new file.

The fourth step involves starting PVM daemons on all allocated machines. PUNCH accomplishes this by invoking the PVM console in the scratch account on the "master" machine; the daemons on the slave machines are automatically started by PVM (via the information in the PVM host file) when the console is invoked.

Once the PVM daemons are running, PUNCH copies the necessary data files into the scratch account on the "master" machine (from the user's PUNCH account) and initiates the PVM-based program (step 5 in the figure). If the program is designed to run in batch mode, it is started in the background; otherwise, it is started within a X-session that is accessible by the user via

his/her browser [4]. A unique feature of the PUNCH approach is that it is a non-intrusive solution — the PVM system and the PVM-based program are completely unaware of PUNCH. One advantage of this is that PUNCH can support unmodified (i.e., legacy) PVM-based programs as long as they do not use hard-coded machine names. (This limitation can be removed by trapping `rsh` calls from PVM and modifying them; see [6] for details.) Another benefit is that PUNCH does not affect the performance of the programs, except to the extent that it makes resource allocation decisions.

At this point, PUNCH simply waits for the PVM-based program to complete (step 6 in Figure 1). When this happens, PUNCH will first retrieve output files from the scratch accounts and place them in the user's PUNCH account. Then, PUNCH will stop the PVM daemons (via the PVM console), terminate any active processes within the allocated scratch accounts, and remove the `.rhost` files. Once PUNCH has verified that the scratch accounts are "clean" (i.e., empty and no active processes), they will be returned to the account pool.

4   Related Work

MPICH-G [7] is a grid-enabled implementation of MPI that uses services provided by the Globus toolkit [8] to allow users to run MPI programs within a wide area network-computing environment. This work makes existing MPI-based programs usable in a network-computing environment by enhancing the capabilities of MPI itself, whereas the PUNCH approach provides support mechanisms that work with unmodified implementations of PVM/MPI. Another difference between the two approaches is that MPICH-G requires users to have user-accounts on all machines that might be used to run the MPI program, whereas PUNCH uses scratch accounts to work around this problem. (PUNCH provides administrators with a way to specify usage policies so that only authorized users are given access to machines.)

Legion [9] allows PVM programs to run in the Legion network-computing environment by emulating the PVM API on top of the Legion run-time system. This approach is fairly complex from an implementation standpoint, and does not support the complete PVM API [10].

Finally, Condor [11] provides support for PVM programs that are based on the master-worker paradigm. One issue that arises in Condor's opportunistic computing environment is that the "master" process must be able to handle the disappearance of worker nodes; the "master" process can compensate for lost nodes by (dynamically) requesting additional machines.

5   Conclusions

A prototype version of PUNCH that allows users to run unmodified PVM-based programs in a wide area network-computing environment has been implemented and tested. Preliminary results show that the described approach efficiently manages available resources. Support for MPI-based programs is being added; this is a relatively simple extension of the work described here.

The implementation described in this paper does not provide support for dynamically increasing or decreasing the number of machines available to a running PVM program. Future work will be aimed at adapting the type and number of machines available to a PVM program on the basis of observed and predicted performance characteristics.

Acknowledgements

References

1. Vaidy Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–547, April 1994.
2. William Gropp, Ewing Lusk, N. Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.
3. Nirav H. Kapadia and José A. B. Fortes. PUNCH: An architecture for web-enabled wide-area network-computing. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 2(2):153–164, September 1999. In special issue on High Performance Distributed Computing.
4. Nirav H. Kapadia, José A. B. Fortes, and Mark S. Lundstrom. The Purdue University Network-Computing Hubs: Running unmodified simulation tools via the WWW. *ACM Transactions on Modeling and Computer Simulation*, 2000. In forthcoming special issue on Web-based Modeling and Simulation.
5. SSH 2.0 protocol specifications. Internet Engineering Task Force (IETF) drafts available at http://info.internet.isi.edu/1/in-drafts.
6. Arash Baratloo, Ayal Itzkovitz, and Zvi M. Kedem. Just-in-time resource management in distributed systems. Technical Report TR1998-762, Department of Computer Science, New York University, March 1998.
7. Ian Foster and Nicholas T. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *Proceedings of the Supercomputing Conference*, 1998.
8. Ian Foster and Carl Kesselman. The Globus project: A status report. In *Proceedings of the 1998 Heterogeneous Computing Workshop (HCW'98)*, pages 4–18, 1998.
9. Andrew S. Grimshaw and William A. Wulf. Legion: Flexible support for wide-area computing. In *Proceedings of the 7th ACM SIGOPS European Workshop*, Ireland, 1996.
10. Roger R. Harper. Interoperability of parallel systems: Running pvm applications in the legion environment. Technical Report CS-95-23, Department of Computer Science, University of Virginia, May 1995.
11. *Condor Version 6.1.12 Manual*, March 2000.

Simulating 2-D Froths; Fingerprinting the Dynamics

H. J. Ruskin* and Y. Feng*+

*School of Computer Applications, Dublin City University, Dublin 9, Ireland.
+Novell Ireland Ltd., Dublin 1, Ireland.

Abstract

Abstract: Topological measures are an obvious choice for investigation of cellular systems, and average topological properties of a froth, defined to be *shell-structured inflatable (SSI)*, have been shown to obey simple relations. However, froth is an intrinsically non-equilibrium system, and *SSI* froths typically become *non-SSI* as coarsening progresses, so that more general probes may provide further insight. Cluster *persistence* permits fingerprinting of froth dynamics at different length scales and facilitates comparison with non-cellular structures. There is evidence to show that the average persistent area in a froth achieves a stable value, but support for power law decay of the average bubble fraction cannot be established for intermediate time scales. We present simulation results for both Voronoi and uniform 2-D froths and examine the case for topological and non-topological probes of the dynamics.

1. Introduction

The soap froth is an ideal model of a cellular network, which is disordered and space-filling, [1-7]. It is an intrinsically *non-equilibrium* system, which evolves to a universal stable state, through surface-energy driven diffusion. Evolution or *coarsening* is associated with *two* separate dynamics, with very different rates of occurrence, [8]. The first is due to rapid topological transformations with corresponding changes in connectivity, which occur system-wide. The second reflects slow, deterministic relaxation over a long time-scale, as a consequence of diffusion of gas between the bubbles. The steady-state evolution of the froth has been characterised by laws describing the statistics of cell area, [9], the growth-rate of $n-$sided cells, [10] and scaling properties of cells [11].

Initially, correlation effects were considered to be restricted to nearest-neighbours only, through the Aboav-Weaire Law [12]. The average number of neighbours of an n-sided cell is given as m(n) = (6-a) + (6a+ μ_2)/n, with μ_2 the second moment of the side distribution *f(n)* and a the Aboav-Weaire parameter. More detailed topological correlations have recently been derived, however, based on analysis of the froth as a system of concentric cells, which can be generated recursively, [13-15]. The distance j between any two cells is the smallest number of edges crossed by paths connecting one to the other. Any cell may be taken as the "germ" cell $j = 0$ and layers or shells of equidistant $j = 0,1,2$ ...cells are such that the jth layer of cells at distance j encloses layers *j-1, j-2.....0* and includes all cells which are themselves neighbours of at least one cell at distance

$j + 1$. Any cell, which does not obey this condition, is said to lie between layers $j - 1$ and j and represents a localised *defect inclusion* with respect to the froth "skeleton". Any froth, without defect inclusions is called *shell-structured inflatable (SSI)*, [13]. In the asymptotic steady-state, topological properties are invariant, with μ_2 achieving a constant value and with the average cell area proportional to the square root of the time. Furthermore, μ_2 is a measure of the disorder in the froth, which affects both the evolution and the fraction of initial cells remaining [16]. These remainder or *survivors* are cells which are present at a given time t_f and which were also present at t_i , $t_i < t_f$, [17]. Most evolutionary properties are based on the contribution of survivors at different stages, so that it is more reasonable to choose a known survivor as the germ cell in a dynamic investigation, although the theory equally applies to any other choice.

Although topological measures are a natural choice for assessing evolution of a froth over time, more general measures provide a useful basis for comparison with systems which do not have cellular structure, [18]. The local decay of persistence towards zero, $P(t_0,t) \sim t^{-\theta}$ was first proposed as a new and general probe of non-equilibrium dynamics [19] and has recently been discussed in some detail. To date, however, numerical simulation results for the value of the exponent θ are not in good agreement with theory and experiment, although a limiting value of $\theta = 1$ is indicated by both, ([18] and refs. therein).

In a froth, the *persistent property* of interest may be taken to be the fraction of the system which has remained within the same bubble from initial time t_0 to given time t. More generally, [20], the known cellular structure of the froth may be exploited for comparison, by definition of a *virtual phase*, where a given fraction ϕ, say, of the bubbles are "coloured" at time t_0 and persistent properties of this fraction are studied as $t \to t_\infty$. The persistent area, is thus bounded above and below respectively, by areas of coloured survivors and ancestors at t_0, [18], (where ancestors are predecessors of the bubbles remaining at time t). Again, selection of known survivors for colouring, facilitates dynamical investigation of the froth properties.

In what follows, we report for both randomly-generated and uniform froths on the dynamics of evolution as charted by both topological and persistence probes.

2. Methodology

Direct simulation, using the method of Weaire and Kermode, [21], provides precise information on independent bubble parameters, with clear distinction made between *topological* and *diffusive* changes. In 2-d, the former include T1 and T2 processes, (side- switching and bubble disappearance respectively), and are effectively instantaneous. Conversely, bubble-size, (number of sides n, area A) evolves continuously, but only cells with $n > 6$ will grow, by von Neumann's Law, with rate dependent on the initial disorder in the froth.

A Voronoi froth is intrinsically disordered and *non-SSI*, (Fig. 1).
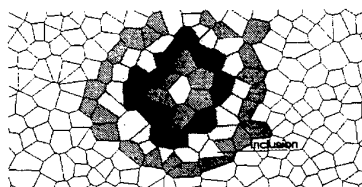
Fig. 1. Voronoi froth, illustrating topological inclusion in shell stucture; intrinsically *non-SSI*

A uniform hexagonal froth however is in mechanical equilibrium, so that bubble movement must be stimulated initially by seeding the froth with one or more *topological dislocations* or *defects*. The simplest forms of defect are achieved by forcing either a T1 or T2 process to give a pentagon-heptagon pairing or an eight-sided single cell respectively. The large cells are obvious survivor choices of germ cell for shell-structure analysis. A single defect has been shown to grow rapidly until it effectively consumes the whole system, [22], and multiple defects expand until impacting with each other, after which changes are slower. Ancestors can be backtracked to the time origin, providing not only a more natural time scale for determining the existence of a fixed distribution, $f(n)$, but a basis to "colour" sensibly the required volume or sampling fraction ϕ in an examination of persistence in the network.
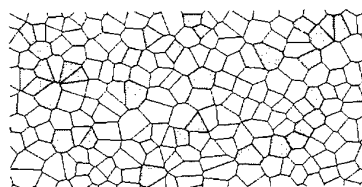
Fig. 2. Voronoi froth, with random colouring for volume (sampling) fraction $\phi \sim 0.2$

The virtual phase for the Voronoi network, (randomly coloured bubbles Fig. 2), and for the hexagonal network, (centred on defect choice, Fig. 3), have been followed over time for different "volume" or sampling fraction, ϕ, ranging from 0.02 to 0.4. This is contrasted, (for the Voronoi), with the behaviour observed if *survivors* at time $t_1 > t_0$ are taken to be the original sample. This latter choice obviously biases the relative area, since bubbles at t_i will be large compared to those at t_0, but ensures that survivors are featured at the crucial period and provides confirmation that the equilibrium value has been achieved. Systems of up to 2500 bubbles were considered.
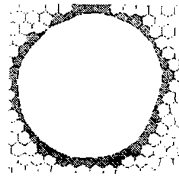
Fig. 3. Hexagonal froth with defect as centrel germ cell choice in shell structure, (SSI). Colouring for volume (sampling) fraction ϕ in persistence also includes at least one such defect.

For the topological measures, we have chosen the germ cell at random for the Voronoi, (as in Fig. 1) and as a defect for the uniform network, (as above). Key equations for the topological properties in an SSI froth have been given, [13] to be

$$K_{j+1} = s_j K_j - K_{j-1} \ (j \geq 1) \tag{1}$$

$$Q_j = 6 - K_{j+1} + K_j \tag{2}$$

where K_j is the total number of cells in the layer j and $s_j = m_{j-4}$ is a constant, (m_j is the average number of sides per cell in the layer j). The logistic map starts with $K_0 = 1$ and $K_1 = n$, the no. of sides (or more generally neighbours) to the central cell. Equation (2) is a special case of the more general expression for *topological charge*, Q_j, from the "Gauss" theorem, [23], where the general form applies to any froth, whether SSI or not.

An approximate expression for the Aboav-Weaire law for higher shell number has been proposed [25] as

$$m_j K_j \approx 6K_j + (2 - a)\mu_2 \tag{3}$$

which is trivial for the second term on the right hand side $= 0$, but suggests that, in the asymptotic limit (for j), a froth can only be free of defects if $\mu_2 = 0$ or $a = 2$ and we have also explored this for the hexagonal network, for controlled disorder.

3. Results

In the Voronoi froth, the number of survivors in the virtual phase at the early time period is large and the distribution of area at t to initial area $(A^*(t)/A(0))$ is left-skewed. However, as the froth coarsens and bubbles disappear, this is gradually reversed, as few survivors have non-zero persistent area. For ϕ very small ~ 0.02, this constitutes a small sampling fraction of the finite system size, (50 in 2500 bubbles), so that over a large number of time steps, the quantity $< A^*(t)/A(0) >$ achieves a relatively stable value of just under 0.4 for the biased sample and this appears to agree reasonably well with the value indicated for the initial random sample, although equilibrium is less clearly established in

this case. As ϕ is increased, (0.1,0.2 say), decay is slower and it is not evident that a time-independent value is finally achieved, although the curves do flatten around $t = 700 - 750$ time steps for the biased sample in both cases. This occurs at a value less than 0.5, and indications are that a similar value is attained for the random sample in both cases. Consequently, the qualitative evidence is reasonably supportive of a time-independent form for $< A^*(t)/A(0) >$ with a value between 0.35 and 0.55, (Fig. 4).
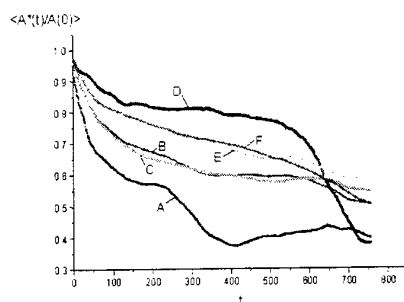
Fig. 4. Average area of persistent regions within a bubble at time t, normalised by area of bubble at time t_0 (persistent area ratio) as a function of time
A (ϕ= 0.02,biased), B(ϕ= 0.1,biased), C(ϕ= 0.2,biased), D(ϕ= 0.02,random), E(ϕ= 0.1,random), F (ϕ= 0.2,random)

Further, $N^*(t)/N(t)$, the fraction of bubbles containing persistent area at time t, is clearly expected to decrease with t and plotted against average bubble area $< A(t) >$ for t large, we might hope to observe decay. Unfortunately, it seems clear that the percentage of initial bubbles which contribute to any decay is extremely small ($< 5\%$) and the simulation time-scale is too short to be able to view this for the Voronoi, with area growth inevitably limited by the finite size of the system. For ϕ very small, there is considerable noise, which decreases as ϕ increases but again no decay can be observed for $N^*(t)/N(t)$.

We have also considered, therefore, the persistence of the virtual phase in a uniform hexagonal network, since the growth in area of the large cell or cells is extremely rapid, so that some contraction of the time-scale may be achieved. The evolution for a single defect is a special case, for which usual asymptotic relations do not apply [22], but as the defect concentration is increased, the evolution of the froth is closer to that of the Voronoi. Again, results for low ϕ are reasonably supportive of a time-independent value for the area ratio, but do depend on whether one or more defects or large cells are included as part of the virtual phase. No decay of $N^*(t)/N(t)$ can be observed for size of systems considered so far (900, 1600 bubbles), although larger hexagonal systems with low defect concentrations and small ϕ merit further study.

With respect to the topological probes, the topological charge is initially constant from the second layer for single defects of both T1- and T2-induced type in the hexagonal froth, with the number of cells in a layer increasing linearly after the first few layers as $K_{j+1} = K_j + 6$ ($j > p$, where p depends on centre cell choice). However, we showed recently that, for the T1-formed froth, inclusions occur very quickly in the first few layers, so that the structure becomes *non-SSI* for the remainder of the evolution. The T2-formed froth, however, remains *dynamically SSI*, with $\mu_2 \neq 0$ only for the zeroth, first and second layers, so that the suggestion that $\mu_2 = 0$ is necessary for a defect-free froth is clearly incorrect. (The second moment μ_2 does not attain a constant value, so Equation (3) clearly does not hold, and further, μ_2 is also slow to stabilise for *low concentration* of defects). A more formal expression, relating two-cell correlators, $\alpha_1(k, n)$, for nearest-neighbours in froth to $n.m(n)$ has been given, [15], and generalised for j and we note that the total number of first neighbours is always known for seeded disorder in the hexagonal structures, so that two-cell correlations may be obtained for the dynamic T2-formed froth, but not in general. For the case of low defect concentration, for example, the percentage of topological inclusions between shell layers is small prior to impact between defects. Nevertheless, inclusions will occur at some stage, so that topological correlations as a function of the layer distance j no longer apply. Although the (single defect), T2-formed, froth is the *only* exception we have found to the general rule for dynamic froths, a *non-SSI* froth with a small percentage of inclusions has statistical distributions similar to those for *SSI* froth and topological properties may still be exploited to some extent. For large amounts of disorder or randomness, more general probes of the non-equilibrium dynamics seem indicated, although choice is limited by the need to reflect the froth's cellular structure.

4. Conclusions

Topological probes arise naturally in soap froth dynamics and shell-structure analysis provides measures, which relate predominantly to *SSI* froths. The (single-defect) T2-formed hexagonal froth is the only example we have found of a *dynamic SSI* froth, which does not require $\mu_2 = 0$ and for which topological relations for charge and two-cell correlation apply directly. For small percentage of inclusions between cell layers, however, *non-SSI* and *SSI* froth have similar statistical distributions. Cluster persistence, on the other hand, provides a *general* probe of non-equilibrium dynamics, but time-scales required to observe persistence decay are very long. Nevertheless, numerical simulation results indicate that time-independent values are achieved for some persistent properties in Voronoi and uniform froths, for a range of sampling fractions ϕ , where persistent area is given roughly to be $< A^*(t)/A(0) > \sim 0.45$ for the former. This indicates the need for further investigation of persistence properties for networks, where the amount of seeded disorder can be controlled.

References

[1] D. Weaire and N. Rivier, Contemp. Phys. 25 (1984), 59

[2] J.A. Glazier. S.P. Gross and J. Stavans, Phys. Rev. A 36 (1987), 306

[3] J. Stavans, Rep. Prog. in Phys. 56 (1993), 733

[4] D. Weaire and M. A. Fortes, Adv. Phys. 43 (1994), 685

[5] B. Derrida, A.J. Bray and C. Godreche, J. Phys A 27 (1994), L357

[6] D. Stauffer, J. Phys. A 27 (1994), 5029

[7] T. Aste and N. Rivier (1997) in Shape Modelling and Applications (IEEE Computer Society Press 1997), p. 2

[8] H. Flyvbjerg, Phys. Rev E 47 (1993), 4037

[9] F.T. Lewis, Anat. Rec. 38 (1928), 341

[10]J. von Neumann , Metal Interfaces 108 (1952), American Society for Metals, Cleveland.

[11]J. Stavans and J.A. Glazier, Phys. Rev. Lett. 62 (1989), 1318

[12]D.A. Aboav, Metallography 3 (1970), 383 : D. Weaire, Metallography 7 (1974), 157

[13]T. Aste, D. Boose and N. Rivier, Phys. Rev. E 54 (1996), 5482

[14]H.M. Ohlenbusch, T. Aste, B. Dubertret and N. Rivier, Eur. Phys. J. B 2 (1998), 211

[15]B. Dubertret, N. Rivier and M.A. Peshkin, J. Phys. A. :Math. Gen. 31 (1998), 879

[16]H. J. Ruskin and Y. Feng, Physica A 247 (1997), 153

[17]B.Levitan and E. Domany, Phys. Rev. E 54 (1996), 2766

[18]W.Y. Tam, A.D. Rutenberg, K. Y. Szeto and B.P. Vollmayr-Lee, J.Phys.; Con densed Matter (1999) preprint

[19]A.J. Bray, B. Derrida and C. Godreche, Europhys. Lett. 27 (1994), 175

[20] B.P. Lee and A.D. Rutenberg, Phys. Rev. Lett. 79, (1997), 4842

[21]D. Weaire and J. P. Kermode, Phil. Mag. B 48 (1983), 245; D. Weaire and J.P. Kermode, Phil. Mag. B 50 (1984), 379; J.P. Kermode and D. Weaire, Comp. Phys. Commun. 60 (1990), 75.

[22]H.J. Ruskin and Y. Feng, J.Phys.: Condensed Matter 7 (1995), L553 .

[23]N. Rivier and T. Aste, Phil. Trans. Roy. Soc. A 354 (1996), 2055

Parallel Branch-and-Bound
for Chemical Engineering Applications:
Load Balancing and Scheduling Issues

Chao-Yang Gau and Mark A. Stadtherr*

Department of Chemical Engineering, 182 Fitzpatrick Hall,
University of Notre Dame, Notre Dame IN 46556, USA

Abstract. Branch-and-prune (BP) and branch-and-bound (BB) techniques are commonly used for intelligent search in finding all solutions, or the optimal solution, within a space of interest. The corresponding binary tree structure provides a natural parallelism allowing concurrent evaluation of subproblems using parallel computing technology. Of special interest here are techniques derived from interval analysis, in particular an interval-Newton/generalized-bisection procedure. In this context, we discuss issues of load balancing and work scheduling that arise in the implementation of parallel BB and BP, and describe and analyze techniques for this purpose. These techniques are applied to solve problems appearing in chemical process engineering using a distributed parallel computing system. Results show that a consistently high efficiency can be achieved in solving nonlinear equations, providing excellent scalability. The effectiveness of the approach used is also demonstrated in the consistent superlinear speedup observed in performing global optimization.

1 Introduction

The continuing success of the chemical and petroleum processing industries depends on the ability to design and operate complex, highly interconnected plants that are profitable and that meet quality, safety, environmental and other standards. Towards this goal, process modeling, simulation and optimization tools are increasingly being used industrially in every step of the design process and in subsequent plant operations. To perform realistic and reliable process simulation and optimization for industrial scale processes, however, requires very large scale computational resources. Parallel computing technology offers the potential to provide the necessary computational power. However, since most currently used problem solving techniques in process modeling and optimization were developed for use on conventional serial machines, it is often necessary to rethink problem solving strategies in order to take full advantage of parallel computing technology.

---

* Author to whom all correspondence should be addressed. Fax: (219) 631-8366; E-mail: markst@nd.edu

In this context, we are particularly interested in the use of parallel computing technology to address reliability issues that arise in solving process engineering problems. The models that must be solved in process simulation problems are typically highly nonlinear and may have multiple solutions. The goal is to find *all* solutions, to insure that the solution or solutions of interest are not missed. Similarly, in optimization problems, the nonlinear programming problems to be solved are typically nonconvex, and there may be several local optima. The goal is to find the *global* optimum, though in some problems finding all of the local optima may be of interest as well. The approach we apply involves the use of interval analysis, combined with branch-and-prune (BP) or branch-and-bound (BB) strategies. Properly implemented, such techniques can find, or more precisely *enclose*, all solutions to a system of nonlinear equations, and can be used to enclose the global optimum, or all local optima, in optimization problems. This can be done with *mathematical and computational certainty*.

Since the subproblems (tree nodes) generated in the tessellation step in BB and BP algorithms are independent, these techniques are particularly amenable to parallel processing. In this paper, we focus specifically on issues of load balancing and scheduling that arise in the implementation of parallel BB and BP, and describe and analyze techniques for this purpose. An application to a problem arising in chemical process engineering is used to demonstrated the effectiveness of the approach used.

2 Distributed Parallel Computing

The solution of realistic, industrial-scale simulation and optimization problems is computationally very intense, and requires the use of adequate computational resources to be done in a timely manner. High performance computing (HPC) technology, in particular parallel computing, provides the computational power to realistically model, simulate, design and optimize complex chemical manufacturing processes. To better use these leading edge technologies in process simulation requires the use of techniques that efficiently exploit parallel computational resources. One of major trends in this regard is the use of distributed computing systems. Typically, in this sort of system, memory is physically distributed, and communication may be done by message passing through some interconnection network.

The use of parallel processing in chemical engineering has attracted significant attention over the past decade or so. There are a variety of applications for which a distributed approach to parallel computing has proven to be effective. In chemical process systems engineering, some examples, that involve either actual implementation on distributed systems, or algorithms appropriate for distributed computing, can be seen in the field of deterministic global optimization and reliable nonlinear equation solving (e.g., [1–9]), nondeterministic global optimization (e.g., [10–12]), BB in process scheduling (e.g., [13–16]), BB in process synthesis (e.g., [10, 17–19]), and process simulation, analysis and optimization

(e.g., [20–39]). There are also a number of important application areas outside of process systems engineering (e.g., [40–45]).

The type of distributed parallel system of particular interest here is a cluster of workstations (COW), in which multiple workstations on a network are used as a single parallel computing resource. This sort of parallel computing system has advantages since it is relatively cheap economically, and is based on widely available hardware. Thus, such an approach to parallel computing has become a important trend in providing high performance computing resources in science and engineering.

3   Branch-and-Bound

Branch-and-prune (BP) and branch-and-bound (BB) algorithms are general-purpose intelligent search techniques for finding all solutions, or the optimal solution, within a space of interest, and have a wide range of applications. These techniques employ successive decomposition (tesselation) of the global problem into smaller disjoint or independent subproblems that are solved recursively until all solutions, or the optimal solution, are found. BB and BP have important applications in engineering and science, especially when a global solution to an optimization problem, or all solutions to a nonlinear equation solving problem are sought. In chemical engineering, these applications include process synthesis (e.g., [10, 17–19]), process scheduling (e.g., [13–16]), analysis of phase behavior (e.g., [46–48]), and molecular modeling (e.g., [49]).

In BP, a subproblem is typically processed in some way to verify the existence of a feasible solution. The subproblem may be examined by a series of tests, and is pruned when it fails specified criteria or if a unique solution can be found inside this subdomain. If no conclusion is available, and so the subproblem cannot be pruned, the problem is bisected into to two additional subproblems (nodes), generating a *binary tree* structure. One of the subproblems is then put in a stack and tests are continued on the other. This type of BP procedure is one of the basic ideas underlying the application of interval analysis to equation-solving problems. More details on interval analysis, in the particular interval-Newton/generalized-bisection (IN/GB) method, are presented in next section. When solving a system of nonlinear equations, the pruning scheme consists of a function range test and the interval-Newton existence and uniqueness test. There are three situations in which an interval (node) can be pruned: (1) zero is not contained in any component of the function range; (2) a unique solution is proven to be enclosed, and (3) it is proven that no solutions exist. With these pruning criteria, a scheme can be constructed that searches the entire binary tree and finds all solutions of the equation system.

In BB, the goal is typically to find a globally optimal solution to some problem. BB may be built on top of BP schemes by enbedding an additional pruning test. In this test, a node is pruned when its optimal (lower bounding) solution is guaranteed to be worse (greater) than some known current best value (an upper bound on the global minimum). Thus, one avoids visiting subproblems which

are known not to contain the globally optimal solution. In this context, various heuristic schemes may be of considerable importance in maintaining search efficiency. For example, when solving global minimization problems using interval analysis, the best upper bound value may be generated and updated by some heuristic combination of an interval extension of the objective function, a point objective function evaluation with interval arithmetic, and a local minimization with a verification by interval analysis. In order to enhance bounding and pruning efficiency, some approaches also apply a priority list scheme in BB. Typically, all problems in the stack are rearranged in the order of some importance index, such as a lower bound value. The idea is that the most important subproblems stored in the stack are examined with higher priority, in the hope that the global optimum be found early in the search process, thus allowing other later subproblems that do not possess the global optimum to be quickly pruned before they generate new nodes.

In BB or BP search, the shape and size of the search space typically changes as the search proceeds. Portions that contain a solution might be highly expanded with many nodes and branches, while portions that have no solutions might be discarded immediately, thus resulting in an irregularly structured search tree. It is only through actual program execution that it becomes apparent how much work is associated with individual subproblems and thus what the actual structure of the search tree is. Since the subproblems to be solved are independent, execution of both BP and BB on parallel computing systems can clearly provide improvements in computational efficiency; thus the use of parallel computing to implement BP and BB has attracted significant attention (e.g., [50–56]). However, because of the irregular structure of the binary tree, this implementation on distributed systems is often not straightforward. Details concerning the methodology for implementing BP and BB on distributed parallel systems will be discussed in later sections.

4   Interval Analysis

Of particular interest here are BP and BB schemes based on interval analysis. A real interval Z is defined as the set of real numbers lying between (and including) given upper and lower bounds; i.e., $Z = [z^L, z^U] = \{z \in \Re \mid z^L \leq z \leq z^U\}$. A real interval vector $\mathbf{Z} = (Z_1, Z_2, \ldots, Z_n)^T$ has n real interval components and can be interpreted geometrically as an n-dimensional rectangle (box). Note that in this section lower case quantities are real numbers and upper case quantities are intervals. Several good introductions to interval analysis are available (e.g., [57–59]). In this section, interval analysis is described in the context of solving nonlinear parameter estimation problems, since that is the primary example used in the tests discussed later. However, it should be emphasized that the interval methods discussed here are general-purpose and can be used in connection with other objective functions in a global optimization problem and other equation systems in an equation solving problem.

BP and BB techniques can be constructed using the interval-Newton technique. Given a nonlinear equation system with a finite number of real roots in some initial interval, this technique provides the capability to find (or, more precisely, narrowly enclose) *all* the roots of the system within the given initial interval. For the unconstrained minimization of an objective function (or estimator) $\phi(\boldsymbol{\theta})$ in parameter estimation, a common approach is to use the gradient of $\phi(\boldsymbol{\theta})$ and seek a solution of $\mathbf{g}(\boldsymbol{\theta}) \equiv \nabla\phi(\boldsymbol{\theta}) = \mathbf{0}$ in order to determine the optimal parameter values $\boldsymbol{\theta}$. The global minimum will be a root of this nonlinear equation system, but there may be many other roots as well, representing local minima and maxima and saddle points. Thus, for this approach to be reliable, the capability to find all the roots of $\mathbf{g}(\boldsymbol{\theta}) = \mathbf{0}$ is needed, and this is provided by the interval-Newton technique. In practice, by using an objective range test, as discussed below, the interval-Newton procedure can also be implemented as a BB technique, so that roots of $\mathbf{g}(\boldsymbol{\theta}) = \mathbf{0}$ that cannot be the global minimum need not be found. The solution algorithm is applied to a sequence of intervals, beginning with some initial interval $\Theta^{(0)}$ specified by the user. This initial interval can be chosen to be sufficiently large to enclose all physically feasible values. It is assumed here that the global optimum will occur at an interior stationary minimum of $\phi(\boldsymbol{\theta})$ and not at the boundaries of $\Theta^{(0)}$. Since the estimator $\phi(\boldsymbol{\theta})$ is derived based on a product of Gaussian distribution functions corresponding to each data point, only a stationary global minimum is reasonable for statistical regression problems such as considered here.

For an interval $\Theta^{(k)}$ in the sequence, the first step in the solution algorithm is the *function range test*. Here an *interval extension* $\mathbf{G}(\Theta^{(k)})$ of the function $\mathbf{g}(\boldsymbol{\theta})$ is calculated. An interval extension provides upper and lower bounds on the range of values that a function may have in a given interval. It is often computed by substituting the given interval into the function and then evaluating the function using interval arithmetic. The interval extension so determined is often wider than the actual range of function values, but it always includes the actual range. If there is any component of the interval extension $\mathbf{G}(\Theta^{(k)})$ that does not contain zero, then we may discard (prune) the current interval (node) $\Theta^{(k)}$, since the range of the function does not include zero anywhere in this interval, and thus no solution of $\mathbf{g}(\boldsymbol{\theta}) = \mathbf{0}$ exists in this interval. We may then proceed to consider the next interval in the sequence, since the current interval cannot contain a stationary point of $\phi(\boldsymbol{\theta})$. Otherwise, if $\mathbf{0} \in \mathbf{G}(\Theta^{(k)})$, then testing of $\Theta^{(k)}$ continues.

The next step is the *objective range test*. The interval extension $\Phi(\Theta^{(k)})$, which contains the range of $\phi(\boldsymbol{\theta})$ over $\Theta^{(k)}$, is computed. If the lower bound of $\Phi(\Theta^{(k)})$ is greater than a known upper bound on the global minimum of $\phi(\boldsymbol{\theta})$, then $\Theta^{(k)}$ cannot contain the global minimum and need not be further tested. Otherwise, testing of $\Theta^{(k)}$ continues. The upper bound on the objective function used for comparison in this step can be determined and updated in a number of different ways. Here we use point evaluations of $\phi(\boldsymbol{\theta})$ done at the midpoint of previously tested Θ intervals that may contain stationary points. Using the objective range test yields a BB procedure for the global minimization of $\phi(\boldsymbol{\theta})$,

while if this step is skipped, we will have a BP technique for finding all solutions of $\mathbf{g}(\boldsymbol{\theta}) = \mathbf{0}$, i.e., all stationary points of $\phi(\boldsymbol{\theta})$.

The next step is the interval-Newton test. Here the linear interval equation system

$$G'(\Theta^{(k)})(\mathbf{N}^{(k)} - \boldsymbol{\theta}^{(k)}) = -\mathbf{g}(\boldsymbol{\theta}^{(k)})$$

is set up and solved for a new interval $\mathbf{N}^{(k)}$, where $G'(\Theta^{(k)})$ is an interval extension of the Jacobian of $\mathbf{g}(\boldsymbol{\theta})$, i.e., the Hessian of $\phi(\boldsymbol{\theta})$, over the current interval $\Theta^{(k)}$, and $\boldsymbol{\theta}^{(k)}$ is a point in the interior of $\Theta^{(k)}$, usually taken to be the midpoint. It has been shown (e.g., [57–59]) that any root $\boldsymbol{\theta}^* \in \Theta^{(k)}$ of $\mathbf{g}(\boldsymbol{\theta}) = \mathbf{0}$ is also contained in the *image* $\mathbf{N}^{(k)}$, implying that if there is no intersection between $\Theta^{(k)}$ and $\mathbf{N}^{(k)}$ then no root exists in $\Theta^{(k)}$, and suggesting the iteration scheme $\Theta^{(k+1)} = \Theta^{(k)} \cap \mathbf{N}^{(k)}$. In addition to this iteration step, which can be used to tightly enclose a solution, it has been proven (e.g., [57–59]) that if $\mathbf{N}^{(k)}$ is contained completely within $\Theta^{(k)}$, then there is *one and only one root* contained within the current interval $\Theta^{(k)}$. This property is quite powerful, as it provides a mathematical guarantee of the existence and uniqueness of a root within an interval when it is satisfied.

There are thus three possible outcomes to the interval-Newton test, as shown schematically for a two variable problem in Figs. 1- 3. The first possible outcome (Fig. 1) is that $\mathbf{N}^{(k)} \subset \Theta^{(k)}$. This represents mathematical proof that there exists a *unique* solution to $\mathbf{g}(\boldsymbol{\theta}) = \mathbf{0}$ within the current interval $\Theta^{(k)}$, and that that solution also lies within the image $\mathbf{N}^{(k)}$. This solution can be rigorously enclosed, with quadratic convergence, by applying the interval-Newton step to the image and repeating a small number of times. Alternatively, convergence to a point approximation of the solution can be guaranteed using a routine point-Newton method starting from anywhere inside of the current interval. Since a unique solution has been identified for this subproblem, it can be pruned, and the next interval in the sequence can now be tested, beginning with the function range test.

The second possible outcome (Fig. 2) is that $\mathbf{N}^{(k)} \cap \Theta^{(k)} = \emptyset$. This provides mathematical proof that no solutions of $\mathbf{g}(\boldsymbol{\theta}) = \mathbf{0}$ exist within the current interval. Thus, the current interval can be pruned and testing of next interval can begin.

The final possible outcome (Fig. 3) is that the image $\mathbf{N}^{(k)}$ lies partially within the current interval $\Theta^{(k)}$. In this case, no conclusions can be made about the number of solutions in the current interval. However, it is known that any solutions that do exist must lie in the intersection $\Theta^{(k)} \cap \mathbf{N}^{(k)}$. If the intersection is sufficiently smaller than the current interval, one can proceed by reapplying the interval Newton test to the intersection. Otherwise, the intersection is bisected, and the resulting two intervals added to the sequence of intervals to be tested. This approach is referred to as an interval-Newton/generalized-bisection (IN/GB) method, and depending on whether or not the objective range test is employed, can be interpreted as either a BB or BP procedure.

It should be emphasized that, when machine computations with interval arithmetic operations are done, the endpoints of an interval are computed with
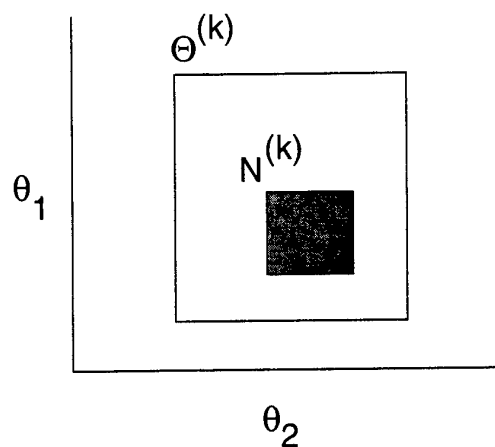
Fig. 1. The computed image $\mathbf{N}^{(k)}$ is a subset of the current interval $\Theta^{(k)}$. This is mathematical proof that there is a unique solution of the equation system in the current interval, and furthermore that this unique solution is also in the image.
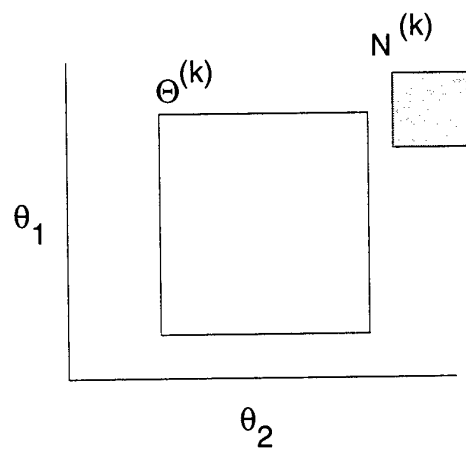
Fig. 2. The computed image $\mathbf{N}^{(k)}$ has a null intersection with the current interval $\Theta^{(k)}$. This is mathematical proof that there is no solution of the equation system in the current interval.
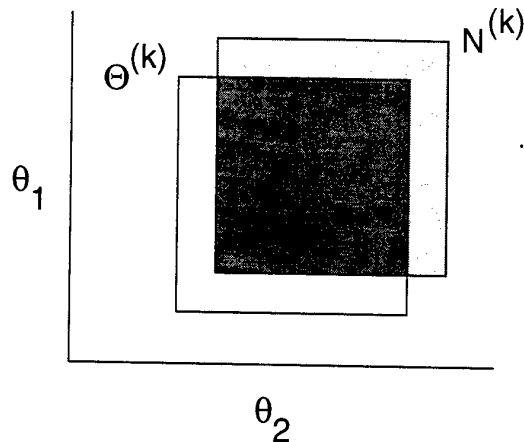
Fig. 3. The computed image $\mathbf{N}^{(k)}$ has a nonnull intersection with the current interval $\Theta^{(k)}$. Any solutions of the equation system must lie in the intersection of the image and the current interval.

a directed outward rounding. That is, the lower endpoint is rounded down to the next machine-representable number and the upper endpoint is rounded up to the next machine-representable number. In this way, through the use of interval, as opposed to floating point arithmetic, any potential rounding error problems are eliminated, yielding an approach that can provide a computational, not just mathematical, guarantee of reliability. Overall, when properly implemented, the IN/GB method described above provides a procedure that is mathematically *and* computationally guaranteed to find the global minimum of $\phi(\boldsymbol{\theta})$, or, if desired, to enclose *all* of its stationary points (within, of course, the specified initial parameter interval $\Theta^{(0)}$).

5 Dynamic Load Balancing and Work Scheduling

As noted above, since the subproblems to be solved are independent, the execution of interval-Newton techniques, whether BP or BB, on distributed parallel systems can clearly provide improvements in computational efficiency. And since, for practical problems, the binary tree that needs to be searched may be quite large, there may in fact be a strong motivation for trying to exploit the opportunity for parallel computing. However, because of the irregular structure of the binary tree, doing this may not be straightforward.

While executing a program to assign the unprocessed workload (stack boxes) to available processors, the irregularity of the tree could cause a highly uneven distribution of work among processors and result in poor utilization of computing resources. Newly generated boxes at some tree nodes, due to bisection, could cause some processors to become highly loaded while others, if processing tree

nodes that can be pruned, could become idle or lightly loaded. In this context, we need an effective dynamic load balancing and work scheduling scheme to perform the parallel tree search efficiently. To manage the load balancing problem, one seeks to apply an optimal work scheduling strategy to transfer workload (boxes to be tested) automatically from heavily loaded processors to lightly loaded processors or processors approaching an idle state. The primary goal of dynamic load balancing algorithms is to schedule workload among processors during program execution, to prevent the appearance of idle processors, while minimizing interprocessor communication cost and thus maximizing the utilization of the computing resources.

A common load balancing strategy is the "manager-worker" scheme (e.g., [3, 4, 7, 12, 19]), in which a single "manager" processor centrally conducts a group of "worker" processors to perform a task concurrently. This scheme has been popular in part because it is relatively easy to implement. It amounts to using a centralized pool to buffer workloads among processors. However, as the number of processors becomes large, such a centralized scheme could result in a significant communication overhead expense, as well as contention on the manager processor. As a result, in many cases, this scheme does not exhibit particularly good scalability. Thus, to avoid bottlenecks and high communication overhead, we concentrate here on decentralized schemes (without a global stack manager), and consider three types of load balancing algorithms specifically designed for network-based parallel computing using message passing.

These parallel algorithms adopt a distributed strategy that allows each processor to locally make workload placement decisions. This strategy helps a processor maintain for itself a moderate local workload stack, hopefully preventing itself from becoming idle, and alleviates bottleneck effects when applied on large-scale multicomputers. All distributed parallel algorithms of this type are basically composed of five phases: workload measurement, state information exchange, transfer initiation, workload placement, and global termination. Each of these phases is now discussed in more detail.

5.1 Workload Measurement

As the first stage in a dynamic load balancing operation, workload measurement involves evaluation of the current local workload using some "work index". This is a criterion that needs to be calculated frequently, and so it must be inexpensive to determine. It also needs to be sufficiently precise for purposes of making good workload placement decisions later. In the context of interval BP and BB, a good approach is to simply use the stack length (number of boxes) as the work index. This index is effective in parallel BP and BB scheme because of the following characteristics:

- A long stack represents a heavy workload and vise versa.
- Exhibiting an empty stack indicates the local processor is approaching an idle state.

- A precise representation of workload by work index may not be needed, since it may not be necessary to maintain an equal workload on all processors, but merely to prevent the appearance of idle states.

Thus, the stack length can serve as a simple, yet effective, workload index.

5.2 State Information Exchange

After all processors identify their own workload state, the parallel algorithm makes this local information available to all other cooperating processors, through interprocessor message passing, to construct a global work index vector. The cooperating processors are a group of processors participating in load balancing operations with a local processor, and define the domain of interprocessor communication, thereby determining a *virtual network* for cooperation. The range of this domain is critical in determining the cost of communication and the performance of load balancing. One possibility is that the cooperating processors could include all processors available on the network, and a global all-to-all communication scheme could then used to update global state information. This provides a very up-to-date global work index vector but might come at the expense of high communication overhead. Alternatively, the cooperating processors might include only a small subset of the available processors, with this small subset defining a local processor's nearest "neighbors" in the virtual network. Now one needs only to employ cheap local point-to-point communication operations. However, without a well-tailored and nested virtual network, and a good load balancing algorithm, these local schemes could result in workload imbalance and idle states.

5.3 Transfer Initiation

After obtaining an overview of the workload state, at least for the group of cooperating ("neighboring") processors, load balancing algorithms now need to decide if a workload placement is necessary to maintain balance and prevent an idle state. This is done according to an initiation policy which dictates under what conditions a workload (box) transfer is initiated, and decides which processors will trigger the load balancing operation. Generally, the migration of boxes from one processor to another processor is initiated on demand. In this context, the load balancing operations are event driven according to different procedures, such as a sender-initiate scheme (e.g., [60–62]), a receiver-initiate scheme (e.g., [63–65]) and a symmetric scheme (e.g., [2, 66, 67]). In the sender-initiate scheme, when the workload of any processor is too heavy and exceeds an upper threshold, the overloaded processor will offload some of its stack boxes to another processor through the network. The receiver-initiate approach works in the opposite way by having an underloaded processor request boxes from heavily loaded processors, when the underloaded processor's workload is less than a lower threshold. The symmetric scheme combines the previous two strategies and allows both underloaded and overloaded processors to initiate load balancing operations.

5.4 Workload Placement

The next step of load balancing algorithm is to complete a workload placement. Here the donor processor splits the local stack into two parts, sending one part to the requesting processor and retaining the other. This operation is done according to a transfer policy consisting of two rules: a work-adjusting rule and a work-selection rule. The work-adjusting rule determinates how to distribute workload among processors and how many stack boxes are to be transferred. If the requesting processor receives too little work, it may quickly become idle; if the donor processor offloads too much work, it itself could also become idle. In either case, the result would eventually intensify the communication needed to perform later load balancing operations. Many approaches are available for this rule. One simple approach is to transfer a constant number of work units (boxes) upon receiving a request, such as in a *work stealing* strategy (e.g. [68]). A more sophisticated approach is to adopt a *diffusive propagation* strategy (e.g. [69–71]), which takes into account the workload states on both sides and adjusts the workload dynamically with a mechanism analogous to heat or mass diffusion.

In addition to the quantity of workload, as measured by the work index, the "quality" of transferred boxes is also an important issue. In this context, a work-selection rule is applied to select the most suitable boxes to transmit in order to supply adequate work to the requesting processor, and thus reduce the demands for further load balancing operations later. Although it is difficult to precisely estimate the size of the tree (or total work) rooted at an unexamined node (box), many heuristic rules have been proposed to select the appropriate boxes. One rule-of-thumb is to transmit boxes near the initial root of the overall binary tree, because these boxes tend to have more future work associated with the subsequent tree rooted at them (e.g., [72]). While this has been demonstrated to be a good selection rule in many tree search applications, this and other such selection rules will not necessarily have a strong influence on the performance of a parallel BP algorithm applied to solve equation-solving problems using interval analysis. However, the selection rule used can have a strong impact on a parallel BB algorithm when solving global minimization problems, since by affecting the evaluation sequence of boxes it in turn affects the time at which good upper bounds on the global minimum are identified. In general, the earlier a good upper bound on the global minimum can be found, the less work that needs to be done to complete the global minimization, since this means it is more likely that boxes can be pruned using an objective range test. This issue will be addressed in more detail in a later section.

5.5 Global Termination

Parallel computation will be terminated when the globally optimal solution for BB problems, or all feasible solutions for BP problems, have been found over the entire binary tree, making all processors idle. For a synchronous parallel algorithm, global termination can be easily detected through global communication

or periodic state information exchange. However, detecting the global termination stage is a more difficult task for an asynchronous distributed algorithm. not only because of the lack of global or centralized control, but also because there is a need to guarantee that upon termination no unexamined workload remains in the communication network due to message passing. One commonly used approach that provides a reliable and robust solution to this problem is Dijkstra's token termination detection algorithm [53, 73, 74].

6 Implementation of Dynamic Load Balancing Algorithms

In this section, a sequence of three algorithms is described for load balancing in a binary tree, with each algorithm in the sequence representing an improvement in principle over the previous one. The last method represents a combination of the most attractive and effective strategies adapted from previous research studies, and also incorporates some novel strategies in this context. Interprocessor communication is performed using the MPI protocol [75, 76], a very powerful and popular technique for massage passing operations that provides various communication functions as discussed below. In the subsequent section, the performance of the three algorithms described will be compared.

6.1 Synchronous Work Stealing (SWS)

This first workload balancing algorithm applies a global strategy, and is illustrated in Fig. 4. All processors are synchronized in the interleaving computation and communication phases. Synchronous blocking all-to-all communication is used to periodically (after some number of tests on boxes) update the global workload state information. Then, every idle processor, if there are any, "steals" one unit of workload (one box) from the processor with the heaviest workload (the largest number of stack boxes), applying a receiver-initiate scheme. As the responsibility for the workload placement decision is given to each individual processor, rather than in a centrally controlling manager processor, but global communication is maintained, SWS can be regarded as a type of *distributed* manager/worker scheme.

The global, all-to-all communication used in this approach provides for an easy determination of workload dynamics, and may lead to a good global load balancing. However, like the centralized manager/worker scheme, this convenience also comes at the expense of increased communication cost when using many processors. Such costs may result in intolerable communication overhead and degradation of overall performance (speedup). It should also be noted that the synchronous and blocking properties of the communication scheme may cause idle states in addition to those that might arise due to an out-of-work condition. When using the synchronous scheme, a processor (sender) that has reached the synchronization point and is ready for communication needs to stay idle and wait for another processor (receiver) to reach the same status, and then initiate
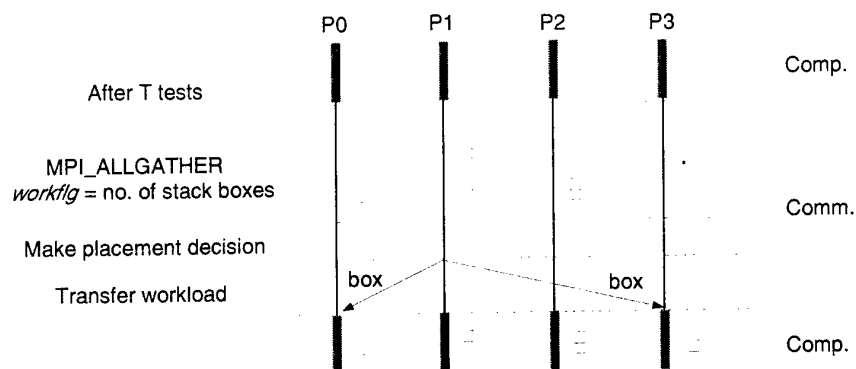
Fig. 4. The SWS algorithm uses global all-to-all communication to synchronize computation and communication phases.

the communication together. Additional waiting states may occur due to the use of blocking communication, since a message-passing operation may not complete and return control to the sending processor until the data has been moved to the receiving processor and a receive posted. Thus, the main difficulties with the SWS approach are the communication overhead and the likely occurrence of idle states, with together may result in poor scalability. However, one advantage to this approach is that the global communication makes it easy to detect global termination.

6.2 Synchronous Diffusive Load Balancing (SDLB)

This second approach for workload balancing follows a localized strategy, by using local, point-to-point communication and a local cooperation strategy in which load balancing operations are limited to a local domain of cooperating processors, i.e., a group of "nearest neighbors" on some predefined *virtual* network. A diffusive work-adjusting rule is also applied here to dynamically coordinate workload transmission between processors, thereby achieving a workload balance with a mechanism analogous to heat or mass diffusion, as illustrated in Fig. 5.

Instead of using global communication, point-to-point synchronous blocking communication is used to exchange workload state information among cooperating (neighbor) processors. The gathered information allows a given processor to construct its own work index vector indicating the workload distribution in its neighborhood. Then, the algorithm uses a symmetric initiation scheme to cause the workload (boxes) to "diffuse" from processors with relatively heavy workloads to processors with relatively light workloads, in order to maintain a roughly equivalent workload over all processors. The virtual network used initially here is simply a ring, which gives each processor two nearest neighbors. Each local processor, i, adjusts its local workload with a neighbor, j, according
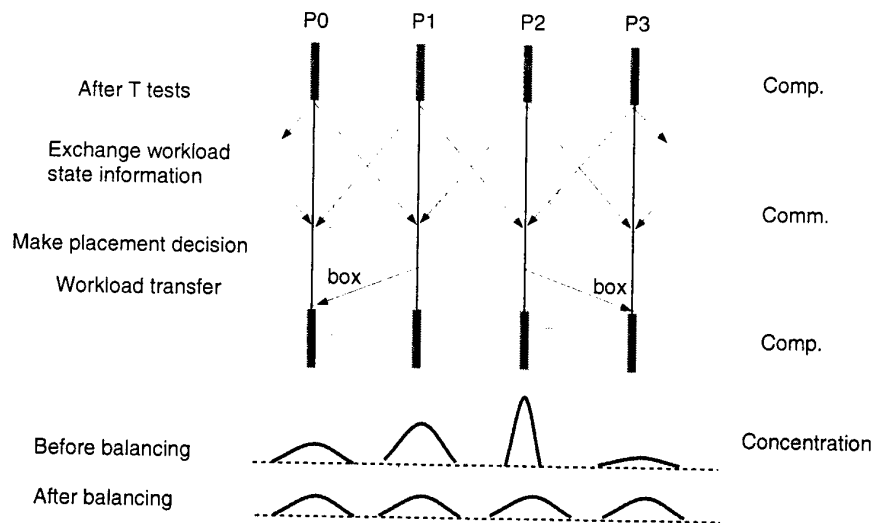
Fig. 5. SDLB uses a diffusive work-adjusting scheme to share workload among neighbors in the virtual network. It is synchronous like SWS.

to the rule

$$u(j) = C[workflg(i) - workflg(j)],$$

where $u(j)$ is the workload-adjusting index, C is a "diffusion coefficient" and $workflg$ is the work index vector. If $u(j)$ is positive and/or greater than a threshold, the local processor sends out workload (boxes); if $u(j)$ is negative and/or less than a threshold, the local processor receives workload (boxes). The diffusion coefficient, C, is a heuristic parameter determining what fraction of local work to offload, and is set at 0.5 in our applications. This diffusive scheme has two advantages. First, when applied at an appropriate frequency, it provides some certainty in preventing the appearance of out-of-work idle states. Also, compacting multiple units of workload (boxes) together for transmission enlarges the *virtual grain* of the transmitted messages. The use of coarse-grained messages to reduce communication frequency tends to minimize the effect of high latency in network transmission, especially on Ethernet. For example, less total time is wasted in startup time of transmission, thus lowering the average transmission cost of a work unit (box), as well as the ratio of communication time to computation time. It should be noted that in considering message grain there may also be maximum message size considerations.

Though the use of a local communication scheme will reduced communication cost to some extent, the use again of synchronous and blocking communication operations are still difficulties in achieving good scalability. On the other hand, while using local rather than global communication makes the detection of global termination less efficient, the synchronous and blocking properties make

this relatively straightforward. Since the problem of detecting global termination becomes more difficult as the number of processors grows, this is another important issue in scalability.

6.3 Asynchronous Diffusive Load Balancing (ADLB)

In this third load balancing approach, a local communication strategy and diffusive work-adjusting scheme are used, as in SDLB. However, a major difference here is the use of an asynchronous nonblocking communication scheme, one of the key capabilities of MPI. The combination of asynchronous communication functionality and nonblocking, persistent communication functionality not only provides for cheaper communication operations by eliminating communication idle states, but also, by breaking process synchronization, makes the sequence of events in the load balancing scheme flexible by allowing *overlap* of communication and computation. As illustrated in Fig. 6., when each processor can perform communication arbitrarily at any time, and independently of a cooperating processor, all communication operations can be scattered among computation, with less time consumed in massage passing.
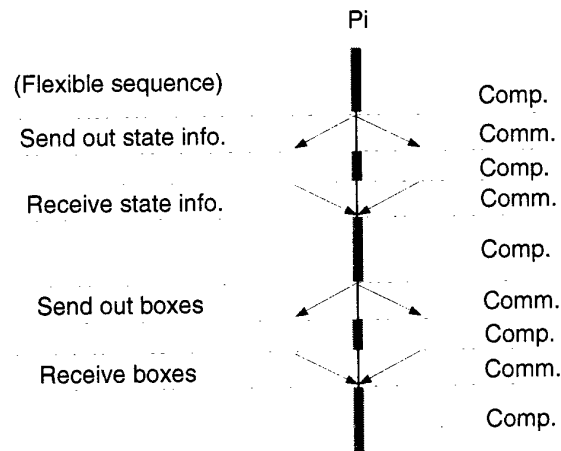
Fig. 6. ADLB uses an asynchronous, nonblocking communication scheme, providing more flexibility to each processor and overlapping communication and computation phases.

In addition to the cheaper and more flexible communication scheme, we incorporate into the ADLB approach two new strategies to try to reduce the demand for communication and thereby try to achieve a higher overall performance. First, as noted above, in BP and BB methods, it is not really necessary to maintain a completely balanced workload across processors. The actual goal

is to prevent the occurrence of idle states by simply maintaining a workload to each processor sufficiently large to keep it busy with computation. To achieve balanced workloads may require a very large number of workload transmissions, resulting in a heavy communication burden. However, in this case, many of the workload transmissions may be unnecessary, since in BP and BB each processor deals with its stack one work unit (box) at a time sequentially, leaving all other workload simply standing by. For a processor to avoid an idle state, and thus have a high efficiency in computation, it is not necessary that its workload be balanced with other processors, but only that it be able to obtain additional workload from another processor through communication as it is approaching an out-of-work state. Thus, we use here a receiver-initiate scheme to initiate work transfer only when the number of boxes in a processor's stack is lower than some threshold, which should be set high enough that the processor is not likely to complete the work and become idle during the processing of workload request to its neighboring processors.

As a consequence, we can also implement a second strategy, which eliminates the periodic state information exchange and combines the load state information of the requesting processor with the workload request message to the donor processor. Upon receiving the request, the donor follows a diffusive work-adjusting scheme as described above for the SDLB approach, but with a modification in the response to the workload adjusting index. Here, if $u(j)$ is positive and/or greater than a threshold, the donor sends out workload (boxes) to the requesting processor; otherwise, it responds that there is no extra workload available. Thus, when approaching idle, a processor sends out a request for work to all its cooperating neighbors, and waits for any processor's donation of work. In case of no work being transferred, it means that the neighbor processors are also starved for work and are making work requests to other neighbors. In this case, the processor will keep requesting work from the same neighbors until they eventually obtain extra work from remote processors and are able to donate parts of it. Through such a diffusive mechanism, heavily loaded processors can propagate workload to lightly loaded processors with a small communication expense.

The last step of this load balancing procedure is to detect global termination. Because the ADLB scheme is asynchronous, the detection of global termination is a more complex issue than in the synchronous case. As noted above, a popular and effective technique for dealing with this issue is Dijkstra's token algorithm [53, 73, 74]. This is the technique used in the ADLB scheme.

In the next section, we describe tests of the three approaches outlined above for load balancing in parallel BP and BB.

7 Computational Experiments and Results

7.1 Test Environment

The performance of an algorithm on a parallel computing system is not only dependent on the problem characteristics and the number of processors but also

on how processors interact with each other, as determined both by a physical architecture in hardware and a virtual architecture in software. The physical architecture used in these tests, as illustrated in Fig. 7, is a network-based system, comprising 16 Sun Ultra 1/140e workstations, physically connected by switched Ethernet. As noted above, in comparison to mainframe systems, such a cluster of workstations (COW) has advantages in its relatively low expense and easy availability of hardware. However, depending on the communication bandwidth and on the communication demands of the algorithm being executed, network contention can have a serious impact on the performance of such a system, particularly if the number of processors is large.
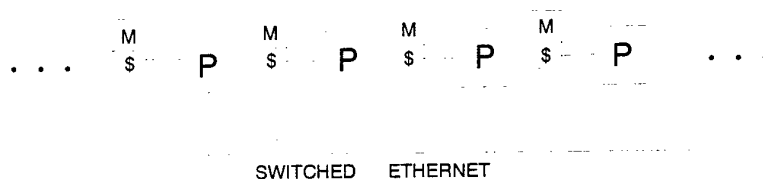
Fig. 7. Physical hardware used is a cluster of workstations connected by switched Ethernet.

Two types of virtual network are used: an all-to-all network (Fig. 8(a)) in the case of SWS, and a one-dimensional torus (ring) network (Fig. 8(b)) in the cases of SDLB and ADLB. In the SWS algorithm, the all-to-all network is implemented by the use of global, all-to-all communication. However, in the SDLB and ADLB algorithms, in order to reduce communication demands and alleviate potential network contention, we only use point-to-point local communication functions and implement the ring network. The load balancing algorithms and test problems were implemented in FORTRAN-77 using the MPI protocol [75, 76] for interprocessor communication.

7.2 Test Problem

The test problem used is a global nonlinear parameter estimation problem involving a vapor-liquid equilibrium (VLE) model (Wilson's equation). Such models, and the estimation of parameters in them, are important in chemical process engineering, since they are the basis for the design, simulation and optimization of widely-used separation processes such as distillation [48]. In this particular problem, we use as the objective function the maximum likelihood estimator, with two unknown standard deviations, to determine two model parameters giving the globally optimal fit of the data to the model [77]. In addition to the difficult nonlinear objective function, the problem data and characteristics were chosen to make this a particularly difficult problem. Interval analysis, as described above, is used to guarantee the correct global solution. The problem can be solved in

(a)   (b)
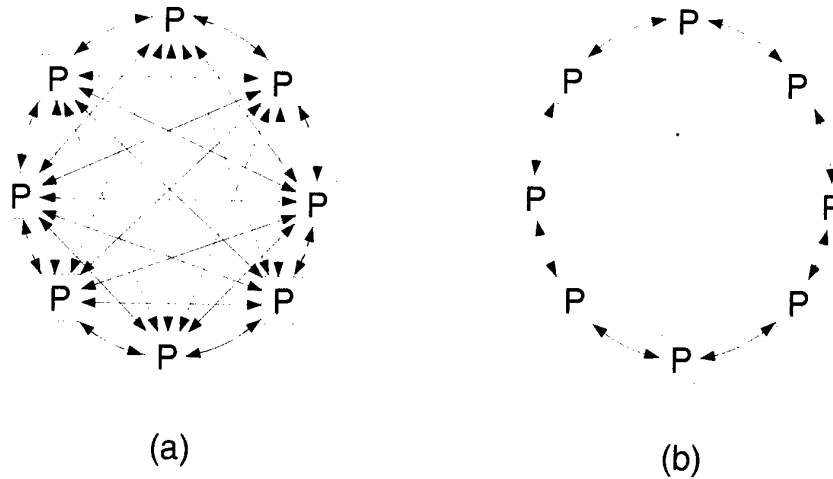
Fig. 8. Virtual network in load balancing: (a) all-to-all network using global communication; used for SWS; (b) 1-D torus network using local communication; used for SDLB and ADLB.

either of two ways. One approach is to treat it as a nonlinear equation solving problem, and use the parallel interval BP algorithm to solve for all stationary points of the objective function (there are five stationary points in this problem). The alternative approach is to treat it directly as a global optimization problem and use the parallel interval BB algorithm. The major difference between the two approaches is the use of the objective range test in the BB algorithm.

7.3 Computational Results

This parameter estimation problem was solved using the COW system described above. During the computational experiments, the COW was dedicated exclusively to solving this problem; that is, there were no other users either on the workstations or on the network. Both the BP scheme solving for all stationary points and the BB scheme merely searching for the global optimum were executed on up to 16 processors using each of the three load balancing schemes described above. Both sequential and parallel execution times were measured in terms of the MPI wall time function, and the performance of each approach evaluated in terms of parallel speedup (ratio of the sequential execution time to the parallel execution time) and parallel efficiency (ratio of the parallel speedup to the number of processors used).

For the interval BP problem of finding all stationary points, the speedups obtained using the three load balancing algorithms, i.e. SWS, SDLB and ADLB, on various number of processors are shown in Fig. 9. All five stationary points were found in every experiment. All points in Fig. 9 are based on an average

over several runs. Since both the sequential runs and all parallel BP runs explored the same binary tree and treated an equivalent amount of total work, the computational results are repeatable and consistent with negligible deviations. As expected, the ADLB approach clearly outperforms SWS and SDLB, exhibiting only slightly sublinear speedup. This can also be seen in the parallel efficiency curves, as shown in Fig. 10. While efficiency curves tend to decrease as the number of processors increases, as a consequence of the Amdahl's law, the ADLB procedure maintains a high efficiency of around 95%. Thus, with the only slightly sublinear speedup and the very high efficiency on up to 16 processors, it seems likely that the ADLB algorithm will be highly scalable to larger numbers of processors.
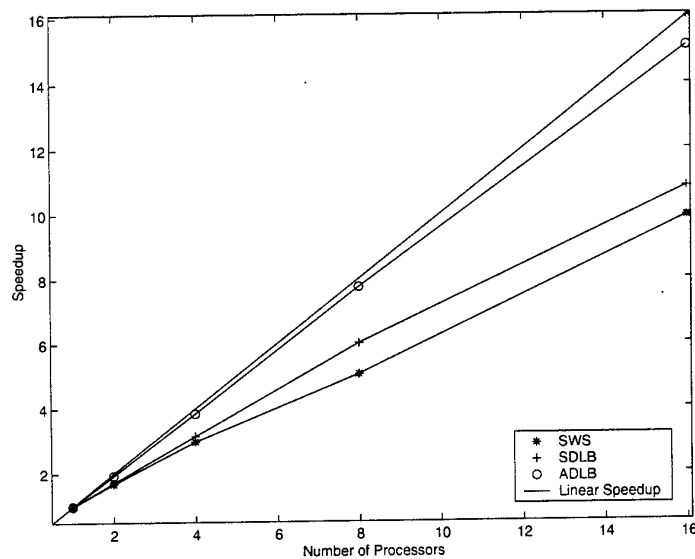
Fig. 9. Comparison of load balancing algorithms on equation solving problem: speedup vs. number of processors.

SWS exhibits the poorest performance of the three load balancing methods. This is partly due to a poor global workload distribution, resulting in a relatively large number of out-of-work idle states, and also partly due to the communication overhead from using the global synchronous blocking communication scheme. In SDLB, the symmetric diffusive work-adjusting scheme using the local communication scheme substantially reduces out-of-work idle states by achieving an even load balance and thus improving the speedup and efficiency. However, while a local communication scheme is employed, the synchronous blocking communication functions used retain a high communication cost and represent a
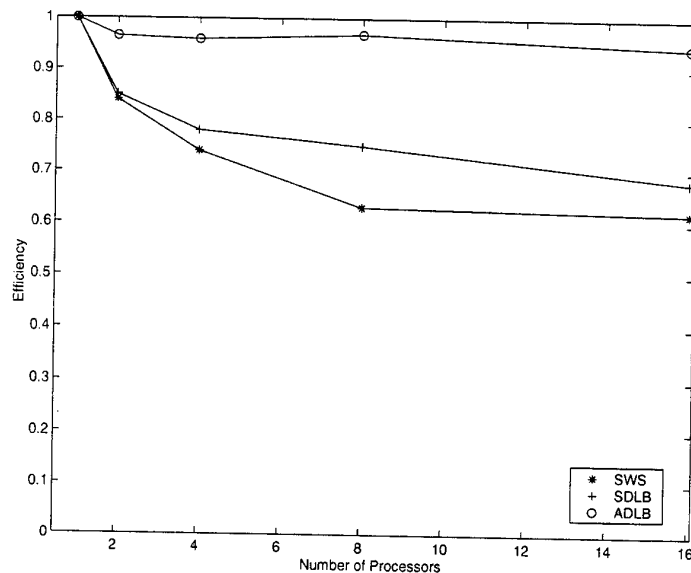
Fig. 10. Comparison of load balancing algorithms on equation solving Problem: efficiency vs. number of processors.

scaling bottleneck. This issue is addressed in ADLB by using asynchronous non-blocking communication functions, allowing the overlap of communication and computation. In addition, by working towards a goal of maintaining non-empty local work stacks instead of an evenly balanced global workload distribution, ADLB provides a large reduction in network communication requirements, thus greatly reducing communication bottlenecks. The reduction of such bottlenecks in ADLB allows it to achieve a consistently high, nearly linear speedup.

For solving the parameter estimation problem as a global optimization problem with parallel interval BB, only the best load balancing scheme, ADLB, was employed. Three different runs using the same problem were made at two, four, eight and 16 processors. The resulting speedups are shown in Fig. 11. We first observe that all speedups are above the linear speedup line, with a speedup over 50 on 16 processors in one case. Superlinear speedup is possible because of the broadcast of least upper bounds, which may cause tree nodes (boxes) to be discarded earlier than in the sequential case, i.e. there is less work to do in the parallel case than in the sequential case. Also, the speedups are not exactly repeatable and may vary significantly from run to run. This occurs because of slightly different timing in finding and broadcasting improved upper bounds in each run. Speedup anomalies, such as the superlinear speedups seen here, are not uncommon in parallel BB search, provided the reduction in the work required in the parallel case (which usually happens but not always) is not outweighed by communication expenses or other overhead in the parallel computation.
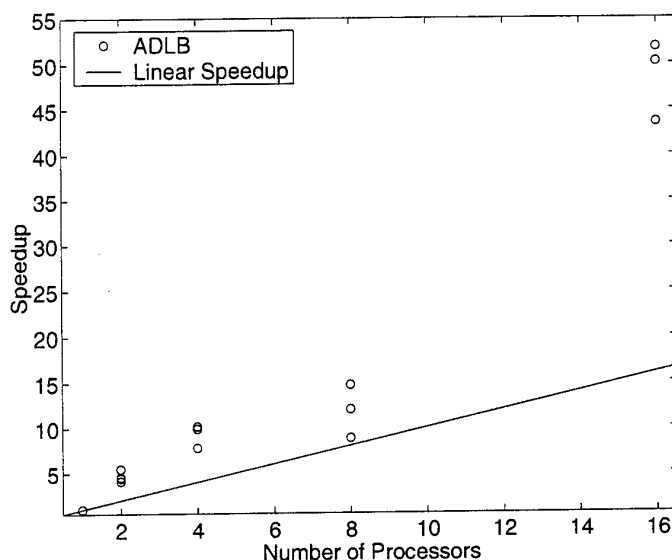
Fig. 11. Speedup anomaly and superlinear speedups are observed in solving the global optimization problem using the parallel BB algorithm based on ADLB.

8   Discussion

The excellent performance of ADLB on the tests described above provides motivation for further improving the ADLB approach for execution on even larger numbers of processors and applied to different sizes of problems. One factor we have investigated is the effect of the underlying virtual network, which is defined to locally coordinate neighbor processors in workload distribution and message propagation. Instead of using a 1-D torus (ring) virtual network, a two dimensional (2-D) torus virtual network, as shown in Fig. 12, has been considered to enhance the load balancing performance. When compared to the 1-D torus, a 2-D torus has a higher communication overhead due to each processor having more neighbors, but it also has a smaller network diameter, $\lceil \sqrt{P}/2 \rceil$ vs. $\lfloor P/2 \rfloor$, thus decreasing the message diffusion distance. It is expected that the trade-off between communication overhead and message diffusion distance may favor the 2-D torus for a larger number of processors.

To evaluate broadly the performance of different parallel algorithms, it is useful to carry out a scalability analysis, which examines how well an algorithm maintains a constant efficiency as the problem size and the number of processors increase. Thus, we carried out an experiment based on the isoefficiency function [53], which determines how much problem size needs to increase in proportion to the number of processors in order to keep the efficiency at a constant level. Small values of the isoefficiency function will correspond to better scalability.
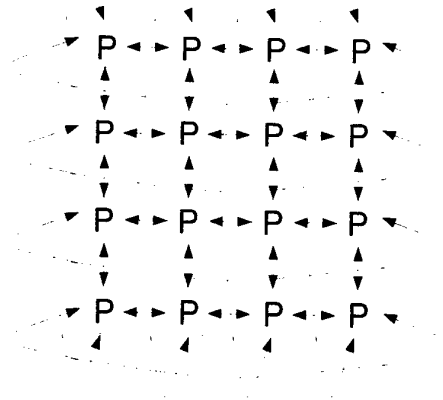
Fig. 12. 2-D torus virtual network is implemented in ADLB to achieve high scalability when running over larger numbers of processors.

We have done preliminary experiments, performing isoefficiency analysis with up to 64 processors, which demonstrate the better scalability of the 2-D torus virtual network on parallel BB and BP problems.

Another issue of interest in this context is how to improve the search efficiency of interval BB for the global optimum. As noted above, there are priority list schemes, such as prioritizing the stack based on a lower bound value, that have been demonstrated to be useful in a variety of branch and bound problems. A difficulty with using lower bound values is that these may not be sufficiently tight to provide any useful heuristic ordering for the evaluation of stack boxes. This is particularly true if the lower bound is obtained by simple interval arithmetic, which often provides only loose bounds when applied to a complicated function.

Thus, we have developed another approach aimed at scheduling the stack boxes for processing. This is a novel dual stack management scheme in which each processor maintains two stacks, a global stack and a local stack. The local stack is unprioritized; that is, with workload appearing in the same sequence as it is generated in the IN/GB algorithm. The local processor draws its work from the local stack as long as it is not empty. This contributes a depth-first pattern to the overall tree search process. The global stack is also unprioritized, and is created by randomly removing boxes from the local stack. The global stack provides boxes for workload transmission to other processors. This contributes breadth to the tree search process. This dual stack management scheme has been demonstrated to be capable of producing consistently high superlinear speedups in BB, reducing the variations from run to run observed previously [78].

9   Concluding Remarks

We have described how load management strategies can be used for effectively solving interval BB and BP problems in parallel on a network-based system. Of the dynamic load balancing algorithms considered, the best performance was achieved by the asynchronous diffusive load balancing (ADLB) approach. This overlaps computation and computation by the use of the asynchronous non-blocking communication functions provided by MPI, and uses a type of diffusive load-adjusting scheme to prevents out-of-work idle states while keeping communication needs small.

The ADLB algorithm was applied in connection with interval analysis, in particular with an interval-Newton/generalized bisection (IN/GB) procedure for reliable nonlinear equation solving and deterministic global optimization. IN/GB provides the capability to find (enclose) all solutions in a nonlinear equation solving problem with mathematical and computational certainty, or the capability to solve global optimization problems with complete certainty. The results of applying ADLB in the equation solving context have shown that the parallel BP algorithm can achieve a nearly linear speedup with a consistently high efficiency around 95% on up to 16 processors in a one-dimensional torus virtual network. Preliminary indications are that ADLB provides high scalability up to 64 processors, and different sizes of problems, when using a 2-D torus virtual network. In the context of global optimization, the parallel BB algorithm achieves significantly superlinear speedups, though is somewhat inconsistent in the extent to which this occurs. By implementing a new dual stack management scheme in connection with ADLB it appears that a consistently high superlinear speedup on optimization problems can be obtained.

Though the test problem here was based on a global parameter estimation problem, it should be emphasized that the parallel IN/GB method is general-purpose and can be used in connection with a wide variety of global optimization problems and nonlinear equation solving problems. Also, the load management schemes described can be applied to a wide variety of other tree search problems in chemical process engineering, such as in process synthesis and process scheduling.

10   Acknowledgments

References

1. Schnepper, C. A., Stadtherr, M. A.: Application of a Parallel Interval Newton/Generalized Bisection Algorithm to Equation-Based Chemical Process Flowsheeting. Interval Comput., **1993**(4) (1993) 40–64

2. Epperly, T. G. W.: Global Optimization of Nonconvex Nonlinear Programs Using Parallel Branch and Bound. Ph.D. thesis. University of Wisconsin, Madison, WI (1995)

3. Androulakis, I.P., Visweswaran, V., Floudas, C. A.: Distributed Decomposition based Approaches in Global Optimization. In: Floudas, C. A., Pardalos, P. M. (eds.): State of the Art in Global Optimization : Computational Methods and Applications, Kluwer Academic Publishers, Dordrecht (1996) 285–301

4. Schnepper, C. A., Stadtherr, M. A.: Robust Process Simulation Using Interval Methods. Comput. Chem. Eng. **20** (1996) 187–199

5. Androulakis, I.P., Floudas, C. A.: Distributed Branch and Bound Algorithms for Global Optimization. In: Pardalos, P. M. (ed.): IMA Volumes in Mathematics and its Applications, Vol. 106. Parallel Processing of Discrete Problems, Springer-Verlag, Berlin (1998) 1–36

6. Berner, S., McKinnon, K. I. M., Millar, C.: A Parallel Algorithm for the Global Optimization of Gibbs Free Energy. Annals of Operations Research **90** (1999) 271–291

7. Smith, E. M. B., Pantelides, C. C.: A Symbolic Reformulation/Spatial Branch-and-Bound Algorithm for the Global Optimization of Nonconvex MINLPs. Comput. Chem. Eng. **23** (1999) 457–478

8. Sinha, M., Achenie, L. E. K., Ostrovsky, G. M.:Parallel Branch and Bound Global Optimizer for Product Design. Presented at AIChE Annual Meeting, Dallas, TX, November (1999).

9. Sinha, M., Achenie, L. E. K.: Parallel Issues in Global Optimization. Presented at AIChE Annual Meeting, Dallas, TX, November, (1999).

10. Acevedo, J., Pistikopoulos, E. N.: Computational Studies of Stochastic Optimization Algorithms for Process Synthesis under Uncertainty. Comput. Chem. Eng. **20** (1996) S1–S6

11. Byrd, R. H., Eskow, E., van der Hoek, A., Schnabel, R. B., Oldenkamp, P. B.: A Parallel Global Optimization Method for Solving Molecular Cluster and Polymer Conformation Problems. Proceedings of the Seventh Siam Conference on Parallel Processing for Scientific Computing, SIAM, Philadelphia, PA (1995) 72–77

12. Kleiber, M., Axmann, J. K.: Evolutionary Algorithms for the Optimization of Modified UNIFAC Parameters. Comput. Chem. Eng. **23** (1998) 63–82

13. Pekny, J. F., Miller, D. L., McRae, G. J.: An Exact Parallel Algorithm for Scheduling When Production Costs Depend on Consecutive System States. Comput. Chem. Eng. **14** (1990) 1009–1023

14. Pekny, J. F., Miller, D. L., Kudva, G. K.: An Exact Algorithm for Resource Constrained Sequencing with Application to Production Scheduling under an Aggregate Deadline. Comput. Chem. Eng. **17** (1993) 671–682

15. Kudva, G. K., Pekny, J. F.: DCABB: A Distributed Control Architecture for Branch and Bound Calculations. Comput. Chem. Eng. **19** (1995) 847–865

16. Subrahmanyam, S., Kudva, G. K., Bassett, M. H., Pekny, J. F.: Application of Distributed Computing to Batch Plant Design and Scheduling. AIChE J. **42** (1996) 1648–161

17. Fraga, E. S., McKinnon, K. I. M.: Process Synthesis Using Parallel Graph Traversal. Comput. Chem. Eng. **18** (1994) S119–S123

18. Fraga, E. S., McKinnon, K. I. M.: The Use of Dynamic Programming with Parallel Computers for Process Synthesis. Comput. Chem. Eng. 18 (1994) 1–13

19. Fraga, E. S., McKinnon, K. I. M.: Portable Code for Process Synthesis Using Workstation Clusters and Distributed Memory Multicomputers. Comput. Chem. Eng. **19** (1995) 759–773

20. Coon, A. B., Stadtherr, M. A.: Parallel Implementation of Sparse LU Decomposition for Chemical Engineering Applications. Comput. Chem. Eng. **13** (1989) 899–914

21. Vegeais, J. A., Stadtherr, M. A.: Parallel Processing Strategies for Chemical Process Flowsheeting. AIChE J. **38** (1992) 1399–1407

22. Zitney, S. E., Stadtherr M. A.: Frontal Algorithms for Equation-Based Chemical Process Flowsheeting on Vector and Parallel Computers. Comput. Chem. Eng. **17** (1993) 319–338

23. Zitney, S. E., Stadtherr, M. A.: Supercomputing Strategies for the Design and Analysis of Complex Separation Systems. Ind. Eng. Chem. Res. **32** (1993) 604–612

24. Beigler, L. T., Tjoa I.-B.: A Parallel Implementation for Parameter Estimation With Implicit Models. Annals of Operations Research **42** (1993) 1–23

25. Moe, H. I., Hertzberg, T.: Advanced Computer Architectures Applied in Dynamic Process Simulation: A Review. Comput. Chem. Eng. **18** (1994) S375–S384

26. O'Neill, A. J., Kaiser, D. J., Stadtherr, M. A.: Strategies for Multicomponent Equilibrium-Stage Separation Calculations on Parallel Computers. AIChE J. **40** (1994) 65–72

27. Coon, A. B., Stadtherr, M. A.: Generalized Block-Tridiagonal Matrix Orderings for Parallel Computation in Process Flowsheeting. Comput. Chem. Eng. **19** (1995) 787–805

28. High, K. A., LaRoche, R. D.: Parallel Nonlinear Optimization Techniques for Chemical Process Design Problems. Comput. Chem. Eng. **19** (1995) 807–825

29. Ingle, N. K., Mountziaris, T. J.: A Multifrontal Algorithm for the Solution of Large Systems of Equations Using Network-Based Parallel Computing. Comput. Chem. Eng. **19** (1995) 807–825

30. Anderson, J. S., Kevrekidis, I. G., Rico-Martinez, R.: A Comparison of Recurrent Training Algorithms for Time Series Analysis and System Identification. Comput. Chem. Eng. **20** (1996) S751–S756

31. Zitney. S. E., Mallya, J. U., Davis, T. A., Stadtherr, M. A.: Multifrontal vs. Frontal Techniques for Chemical Process Simulation on Supercomputers. Comput. Chem. Eng. **20** (1996) 641–646

32. Mallya, J. U., Stadtherr, M. A.: A Multifrontal Approach for Simulating Equilibrium-Stage Processes on Supercomputers. Ind. Eng. Chem. Res. **36** (1997) 144–151

33. Mallya, J. U., Zitney, S. E., Choudhary, S., Stadtherr, M. A.: A Parallel Frontal Solver for Large Scale Process Simulation and Optimization. AIChE J. **43** (1997) 1032–1040

34. Paloschi J. R.: Steps towards Steady-State Simulation on MIMD machines: Solving Almost Block Diagonal Linear Systems. Comput. Chem. Eng. **21** (1997) 691–701

35. Abdel-Jabbar, N., Carnahan, B., Kravaris, C.: A Multirate Parallel-Modular Algorithm for Dynamic Process Simulation Using Distributed Memory Multicomputers. Comput. Chem. Eng. **23** (1999) 733–761

36. Mallya, J. U., Zitney, S. E., Choudhary, S., Stadtherr, M. A.: Matrix Reordering Effects on a Parallel Frontal Solver for Large Scale Process Simulation. Comput. Chem. Eng. **23** (1999) 585–593

37. Vazquez, G. E., Ponzoni, I., Brignole, N. B.: Parallel Depth-First Search on Clusters of Workstations. Presented at SIAM Annual Meeting, Atlanta, GA, May 12-15 (1999)

38. Vazquez, G. E., Brignole, N. B.: Parallel Distributed Optimization for Chemical Engineering Applications. Presented at SIAM Annual Meeting, Atlanta, GA, May 12-15 (1999)
39. Brignole, N. B., Ponzoni, I., Sanchez, M. C., Vazquez, G. E.: A Parallel Algorithm for Observability Analysis on Networks of Workstations. Presented at AIChE Annual Meeting, Dallas, TX, November (1999)
40. Keunings, R.: Parallel Finite Element Algorithms Applied to Computational Rheology. Comput. Chem. Eng. **19** (1995) 647-669
41. Killough, J. E.: The Application of Parallel Computing to the Flow of Fluids in Porous Media. Comput. Chem. Eng. **19** (1995) 775-786
42. Stark, S. M., Neurock, M., Klein, M. T.: Comparison of MIMD and SIMD Strategies for Monte Carlo Modeling of Kinetically Coupled Reactions. Comput. Chem. Eng. **19** (1995) 719-742
43. Traenkle, F., Hill, M. D., Kim, S.: Solving Microstructure Electrostatics on a Proposed Parallel Computer. Comput. Chem. Eng. **19** (1995) 743-757
44. Yang, H., Kim, S.: Boundary Element Analysis of Particle Mobility in a Cylindrical Channel: Network-Based Parallel Computing with Condor. Comput. Chem. Eng. **19** (1995) 683-692
45. Poulain, C. A., Finlayson, B. A.: Distributed Computing with Personal Computers. AIChE J. **42** (1996) 290-295
46. McDonald, C.M., Floudas, C. A.: GLOPEQ: A New Computational Tool for the Phase and Chemical Equilibrium Problem. Comput. Chem. Eng. **21** (1997) 1-23
47. Hua, J. Z., Brennecke, J. F., Stadtherr, M. A.: Enhanced Interval Analysis for Phase Stability: Cubic Equation of State Models. Ind. Eng. Chem. Res. **37** (1998) 1519-1527
48. Gau, C.-Y., Brennecke, J. F., Stadtherr, M. A.: Reliable Nonlinear Parameter Estimation in VLE Modeling. Fluid Phase Equilib. **168** (2000) 1-18.
49. Maranas, C. D., Floudas, C. A: Global Optimization for Molecular Conformation Problems. Anns. Opns. Res. **42** (1993) 85-117
50. McKeown, G. P., Rayward-Smith, V. J., Rush, S. A.: Parallel Branch-and-Bound. In: Kronsj, L., Shumsheruddin, D. (eds.): Advances in Parallel Algorithms, Halsted Press, New York (1992) 111-150
51. Rushmeier, R. A.: Experiments with Parallel Branch-and-Bound Algorithms for the Set Covering Problems. Oper. Res. Lett. **13** (1993) 277-285
52. Gendron, B., Crainic, T. G.: Parallel Branch-and-Bound Algorithms - Survey and Synthesis. Oper. Res. **42** (1994) 1042-1066
53. Kumar. V., Grama, A., Gupta, A., Karypis, G.: Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms. Benjamin-Cummings, Redwood City, CA (1994)
54. Correa, R., Ferreira, A.: Parallel Best-First Branch-and-Bound in Discrete Optimization: A Framework. In: Ferreira, A., Pardalos, P. (eds.): Solving Combinatorial Optimization Problems in Parallel. Vol 1054, Springer-Verlag, Berlin (1996) 171-200
55. Mitra, G., Hai, I., Hajian, M. T.: A Distributed Processing Algorithm for Solving Integer Problems Using a Cluster of Workstations. Parallel Computing **23** (1997) 733-753
56. Correa, R. C.: A Parallel Approximation Scheme for the Multiprocessor Scheduling Problem. Parallel Computing **26** (2000) 47-72
57. Neumaier, A.: Interval Methods for Systems of Equations. Cambridge University Press, Cambridge, UK (1990)

58. Hansen, E. R.: Global Optimization Using Interval Analysis. Marcel Dekkar, New York, NY (1992)
59. Kearfott, R. B.: Rigorous Global Search: Continuous Problems, Kluwer Academic Publishers, Dordrecht (1996)
60. Vornberger, O.: Load Balancing in a Network of Transputers. Second International Workshop on Distributed Algorithms, Lecture Notes in Computer Science, Vol. 312 (1987) 116-126
61. Troya, J., Ortega, M.: A Study of Parallel Branch-and-Bound Algorithms with Best-Bound-First Search. Paral. Comput. **11** (1989) 121-126
62. Quinn, M. J.: Analysis and Implementation of Branch-and-Bound Algorithms on a Hypercube Multicomputer. IEEE Trans. Comp. **39** (1990) 384-387
63. Vornberger, O.: Implementing Branch-and-Bound in a Ring of Processors. Proceedings of CONPAR 86 Conference on Algorithms and Hardware for Parallel Processing, Lecture Notes in Computer Science, Vol. 237 (1986) 157-164
64. Finkel, R., Manber, U.: DIB- A Distributed Implementation of Backtracking. ACM Tran. Prog. Lang. and Syst. **9** (1987) 235-256
65. Nageshwara Rao, V., Kumar, Parallel Depth First Search. 1. Implementation. Int. J. Paral. Prog. **16** (1987) 479-499
66. Luling, R., Monien, B.: Two strategies for Solving the Vertex Cover Problem on a Transputer Network. Third International Workshop on Distributed Algorithms, Lecture Notes in Computer Science, Vol. 392 (1989) 160-170
67. Clausen, J., Traff, J. L.: Implementations of Parallel Branch-and-Bound Algorithms - Experience with the Graph Partitioning Problem. Anns. Opns. Res. **33** (1991) 331-349
68. Blumofe, R. D., Leiserson, C. E.: Scheduling Multithreaded Computations by Work Stealing. In Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS), Santa Fe, New Mexico, November (1994) 356-368
69. Cybenko, G.: Dynamic Load Balancing for Distributed Memory Multiprocessors. J. Parallel Distr. Com. **7** (1989) 278-301
70. Heirich, A., Taylor, S.: Load Balancing by Diffusion. Proceedings of 24th International Conference on Parallel Programming, Vol. 3, CRC Press (1995) 192-202
71. Heirich, A., Arvo, J.: A Competitive Analysis of Load Balancing Strategies for Parallel Ray Tracing. J. Supercomputing **12** (1998) 57-68
72. Foster, I.: Designing and Building Parallel Programs - Concepts and Tools for Parallel Software Engineering. Addison-Wesley, Reading, MA (1995)
73. Dijkstra, E. W., Scholten, C. S.: Termination Detection for Diffusing Computations. Inf. Process. Lett. **11** (1980) 1-4
74. Dijkstra, E. W., Feijen, W. H., van Gasteren, A. J. M..: Derivation of a Termination Detection Algorithm for Distributed Computations. Inf. Process. Lett. **16** (1983) 217-219
75. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge, MA (1994)
76. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced Features of the Message-Passing Interface. MIT Press, Cambridge, MA (1999)
77. Gau, C.-Y., Stadtherr, M. A.: Global Nonlinear Parameter Estimation Using Interval Analysis: Parallel Computing Strategies. Presented at AIChE Annual Meeting, Miami Beach, FL, November (1998)
78. Gau, C.-Y., Stadtherr, M. A.: A Systematic Analysis of Dynamic Load Balancing Strategies for Parallel Interval Analysis. Presented at AIChE Annual Meeting, Dallas, TX, November (1999).

A Parallel Implementation of an Interior-Point Algorithm for Multicommodity Network Flows [*]

Jordi Castro[1, **] and Antonio Frangioni[2]

[1] Statistics and Operations Research Dept., Universitat Politècnica de Catalunya
Pau Gargallo 5, 08028 Barcelona (Spain) `jcastro@eio.upc.es`
[2] Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa (Italy) `frangio@di.unipi.it`

Abstract. A parallel implementation of the specialized interior-point algorithm for multicommodity network flows introduced in [5] is presented. In this algorithm, the positive definite systems of each iteration are solved through a scheme that combines direct factorization and a preconditioned conjugate gradient (PCG) method. Since the solution of at least k independent linear systems is required at each iteration of the PCG, k being the number of commodities, a coarse-grained parallellization of the algorithm naturally arises. Also, several other minor steps of the algorithm are easily parallelized by commodity. An extensive set of computational results on a shared memory machine is presented, using problems of up to 2.5 million variables and 260,000 constraints. The results show that the approach is especially competitive on large, difficult multicommodity flow problems.

1 Introduction

Multicommodity flows are among the most challenging linear problems, due to the large size of these models in real world applications (e.g., routing in telecommunications networks). Indeed, these problems have been used to test the efficiency of early interior-point solvers for linear programming [1]. The need to solve very large instances has led to the development of both specialized algorithms and parallel implementations.

In this paper, we present a parallel implementation of a specialized interior-point algorithm for multicommodity flows [5]. In this approach, the block-angular structure of the coefficient matrix is exploited for performing in parallel the solution of small linear systems related to the different commodities, unlike general-purpose parallel interior-point codes [2, 8, 17] where the parallelization effort is focused on the Cholesky factorization of one large system. This has already been proposed [16, 9, 13]; however, all the previous approaches require to compute and factorize the Schur complement. This can become a significant serial bottleneck,

---

2   J. Castro and A. Frangioni

since this matrix is usually prohibitively dense. Although this bottleneck can be partly eluded by using parallel linear algebra routines, our approach takes a more radical route by avoiding to form the Schur complement, and using an iterative method instead. There have been other proposals along these lines [22, 14], but limited to the sequential case; also, so far no results have been shown for these algorithms. The implementation presented in this paper significantly improves on the preliminary one described in [6]. There, only some of the major routines were parallelized, and less attention was paid to communication and data distribution. Working on these details allowed us to obtain new and better computational results.

From the multicommodity point of view, this approach differentiates itself from most other parallel solvers [7, 15, 19, 25, 21, 12] in that is not based on a decomposition approach. The structure of the multicommodity flow problem has led to a number of specialized algorithms, most of which share the idea of decomposing in some way the problem into a set of smaller independent problems. These are all iterative methods, where at each step the subproblems are solved, and their results are used in some way to modify the subproblems to be solved at the next iteration. Hence, these approaches are naturally suited for coarse-grained parallelization. Parallel price-directive decomposition approaches have been proposed based on bundle methods [7, 19], analytic center methods [12] or linear-quadratic penalty functions [21]. Parallel resource-directive approaches are described in [15]. Finally, experiences with a parallel interior-point decomposition method are presented in [25]. A discussion of these and other parallel decomposition approaches can be found in [7]. A general description of the parallelization of mathematical programming algorithms can be found in [3, 23].

The paper is organized as follows. Section 2 presents the formulation of the problem to be solved. Section 3 outlines the specialized interior-point algorithm for multicommodity flows proposed in [5], including a brief description of the general path-following method. Section 4 deals with the parallelization issues of the algorithm. Finally, Section 5 presents and discuss the computational results.

2 Problem Formulation

The multicommodity flow problem requires to find the least-cost routing of a set of k commodities through a network of m nodes and n arcs, where the arcs have an individual capacity for each commodity, and a mutual capacity for all the commodities. The node-arc formulation of the problem is

$$
\min \sum_{i=1}^{k} c^i x^i
$$

$$
\text{s.t.} \quad
\begin{bmatrix}
E & 0 & \dots & 0 & 0 \\
0 & E & \dots & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \dots & E & 0 \\
\hline
I & I & \dots & I & I
\end{bmatrix}
\begin{bmatrix}
x^1 \\
x^2 \\
\vdots \\
x^k \\
\hline
x^0
\end{bmatrix}
=
\begin{bmatrix}
b^1 \\
b^2 \\
\vdots \\
b^k \\
\hline
u
\end{bmatrix}
\tag{1}
$$

$$
0 \le x^0 \le u , \qquad 0 \le x^i \le u^i \quad i = 1 \dots k.
$$

Vectors $x^i \in \mathbb{R}^n$ are the flow arrays for each commodity, while $x^0 \in \mathbb{R}^n$ are the slacks of the mutual capacity constraints. $E \in \mathbb{R}^{m \times n}$ is the node-arc incidence matrix of the underlying directed graph, while I denotes the $n \times n$ identity matrix. We shall assume that E is a full row-rank matrix: this can always be guaranteed by removing any of the redundant node balance constraints. $c^i \in \mathbb{R}^n$ and $u^i \in \mathbb{R}^n$ are respectively the flow cost vector and the individual capacity vector for commodity i, while $u \in \mathbb{R}^n$ is the vector of the mutual capacities. Finally, $b^i \in \mathbb{R}^m$ is the vector of supplies/demands for commodity i at the nodes of the network.

The multicommodity flow problem is a linear program with $\bar{m} = km + n$ constraints and $\bar{n} = (k+1)n$ variables. In some real-world models, k can be very large: for instance, in many telecommunication problems a commodity represents the flow of data/voice between two given nodes of the network, and therefore k is $O(m^2)$. Thus, the resulting linear program can be huge even for graphs of moderate size. However, the coefficient matrix of the problem is highly structured: it has a block-staircase form, each block being a node-arc incidence matrix. Several methods have been proposed which exploit this structure; one is the specialized interior-point algorithm to be described in the next paragraph.

3   A Specialized Interior-Point Algorithm

In [5], a specialized interior-point algorithm for multicommodity flows has been presented and tested. This algorithm, and the code that implements it, will be referred to as IPM.

IPM is a specialization of the path-following algorithm for linear programming [26]. Let us consider the following linear programming problem in primal form

$$\min \{ \, cx \ : \ Ax = b, \ x + s = u, \ x, s \geq 0 \, \} \, , \tag{2}$$

where $x \in \mathbb{R}^{\bar{n}}$ and $s \in \mathbb{R}^{\bar{n}}$ are respectively the primal variables and the slacks of the box constraints, $u \in \mathbb{R}^{\bar{n}}$, $c \in \mathbb{R}^{\bar{n}}$ and $b \in \mathbb{R}^{\bar{m}}$ are respectively the upper bounds, the cost vector and the right hand side vector, and $A \in \mathbb{R}^{\bar{m} \times \bar{n}}$ is a full row-rank matrix. The dual of (2) is

$$\min \{ \, yb - wu \ : \ yA + z - w = c, \ z, w \geq 0 \, \} \, , \tag{3}$$

where $y \in \mathbb{R}^{\bar{m}}$, $z \in \mathbb{R}^{\bar{n}}$ and $w \in \mathbb{R}^{\bar{n}}$ are respectively the dual variables of the structural constraints $Ax = b$, the dual slacks and the dual variables of the box constraints $x \leq u$.

Replacing the inequalities in (2) by a logarithmic barrier in the objective function, with parameter μ, the KKT optimality conditions of the resulting problem are

$$
\begin{aligned}
r_{xz} &\equiv \mu e - XZe = 0 \\
r_{sw} &\equiv \mu e - SWe = 0 \\
r_b &\equiv \ b - Ax = 0 \\
r_c &\equiv \ c - (yA + z - w) = 0 \\
r_u &\equiv \ u - x - s = 0 \\
&\quad (x, s, z, w) \geq 0 \, ,
\end{aligned}
\tag{4}
$$

4   J. Castro and A. Frangioni

where e is the vector of 1's of proper dimension, and each uppercase letter corresponds to the diagonal matrix having as diagonal elements the entries of the corresponding lowercase vector. In the algorithm we impose $r_u = 0$, i.e. $s = u - x$, thus eliminating $\bar{n}$ variables.

The (unique) solutions of (4) for each possible $\mu > 0$ describe a continuous trajectory, known as the *central path*, which, as μ tends to 0, converges to the optimal solutions of (2) and (3). A path-following algorithm attempts to reach close to these optimal solutions by following the central path. This is done by performing a damped version of Newton's iteration applied to the nonlinear system (4), as shown in (5). A more detailed description of the algorithm can be found in many linear programming textbooks, e.g. [26].

$$
\begin{aligned}
&\textbf{Procedure } PathFollowing(A, b, c, u): \\
&\quad \text{Initialize } x > 0, s > 0, y, z > 0, w > 0; \\
&\quad \textbf{while } (x, s, y, z, w) \text{ is not optimum } \textbf{do} \\
&\qquad \Theta = (X^{-1}Z + S^{-1}W)^{-1}; \\
&\qquad r = S^{-1}r_{sw} + r_c - X^{-1}r_{xz}; \\
&\qquad (A\Theta A^T)\Delta y = r_b + A\Theta r; \\
&\qquad \Delta x = \Theta(A^T \Delta y - r); \\
&\qquad \Delta w = S^{-1}(r_{sw} + W\Delta x); \\
&\qquad \Delta z = r_c + \Delta w - A^T \Delta y; \\
&\qquad \text{Compute } \alpha_P >, \alpha_D > 0; \\
&\qquad x \leftarrow x + \alpha_P \Delta x; \\
&\qquad (y, z, w) \leftarrow (y, z, w) + \alpha_D(\Delta y, \Delta z, \Delta w);
\end{aligned}
\tag{5}
$$

The main computational burden of the algorithm is the solution of the system

$$
(A\Theta A^T)\Delta y = r_b + A\Theta r \equiv \bar{b} .
\tag{6}
$$

Note that $A\Theta A^T$ is symmetric and positive definite, as Θ is clearly a positive definite diagonal matrix. Usually, interior-point codes solve (6) through a Cholesky factorization, preceeded by a permutation of the columns of A aimed at minimizing the fill-in effect. Several effective heuristics have been developed for computing such a permutation. Unfortunately, when A is the constraints matrix of (1), the Cholesky factors of $A\Theta A^T$ turn out to be rather dense anyway [5].

However, the structure of A can be used to solve (6) without computing the factorization of $A\Theta A^T$. Note that Θ is partitioned into the k blocks Θ^i, $i = 1 \ldots k$, one for each commodity, plus the block Θ^0 corresponding to the slack variables x^0 of the mutual capacity constraints. Hence,

$$
A\Theta A^T = \left[\begin{array}{c|c} B & C \\ \hline C^T & D \end{array} \right] = \left[\begin{array}{ccc|c} E\Theta^1 E^T \ldots & 0 & E\Theta^1 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \ldots E\Theta^k E^T & E\Theta^k \\ \hline \Theta^1 E & \ldots \Theta^k E & \Theta^0 + \sum_{i=1}^k \Theta^i \end{array} \right]
\tag{7}
$$

i.e., B is the block diagonal matrix having the $m \times m$ matrices $B_i = E\Theta^i E^T$, $i = 1 \ldots k$, as diagonal elements, and

$$C^T = \begin{bmatrix} C_1^T \ldots C_k^T \end{bmatrix} = \begin{bmatrix} \Theta^1 E \ldots \Theta^k E \end{bmatrix} .$$

Exploiting (7), and partitioning the vectors Δy and $\bar{b}$ accordingly, the solution of (6) is reduced to

$$\left(D - \sum_{i=1}^{k} C_i^T B_i^{-1} C_i \right) \Delta y^0 = \bar{b}^0 - \sum_{i=1}^{k} C_i^T B_i^{-1} \bar{b}^i \equiv \beta^0 \tag{8}$$

$$B_i \Delta y^i = (\bar{b}^i - C_i \Delta y^0) \equiv \beta^i, \qquad i = 1 \ldots k . \tag{9}$$

The matrix

$$H = D - C^T B^{-1} C = D - \sum_{i=1}^{k} C_i^T B_i^{-1} C_i \tag{10}$$

is known as the Schur complement.

Thus, (6) can be solved by means of (8), involving the Schur complement H, followed by the k subsystems (9) involving the matrices B_i. The latter step can be easily parallelized. However, solving (8) with a direct method, as advocated in [16,9], requires forming and factorizing H. As shown in [5], this matrix typically becomes rather dense, hence such a direct approach may become computationally too expensive. Furthermore, it represents a formidable serial bottleneck for a parallel implementation of the code. As suggested in [16], this bottleneck can be reduced by using parallel linear algebra routines [2,8,17]. However, it is also possible to avoid forming H at all, solving (9) by means of an iterative algorithm.

Since H is symmetric and positive definite, a preconditioned conjugate gradient (PCG) method can be used. In [5], a family of preconditioners is proposed, based on the following characterization of the inverse of H:

$$H^{-1} = \left(\sum_{i=0}^{\infty} (D^{-1}Q)^i \right) D^{-1} \qquad \text{where} \qquad Q = \sum_{i=1}^{k} C_i^T B_i^{-1} C_i \tag{11}$$

A preconditioner for (9) can be obtained by truncating the above power series at the h-th term. Clearly, the higher h, the better the preconditioning will be, and the fewer PCG iterations will be required. However, preconditioning one vector requires solving $k \times h$ linear systems involving the matrices B_i, thereby increasing the cost of each PCG iteration. The best trade-off between the reduction of the iterations count and the cost of each iteration is $h = 0$, corresponding to the diagonal preconditioner D^{-1} [5].

The IPM code, implementing this algorithm, has shown to be competitive with a number of other sequential approaches [5]. It is written mainly in C, with only the Cholesky factorization routines (devised by E. Ng and B. Peyton [20]) coded in Fortran. Both the sequential and parallel versions can be freely obtained for academic purposes from

http://www-eio.upc.es/ jcastro/software.html.

6   J. Castro and A. Frangioni

4 Parallelization of the Algorithm

The solution of (6) is by far the most expensive procedure in the interior-point algorithm, consuming up to 97% of the total execution time for large problems. With the above approach, this can be accomplished by means of the following steps:

- Factorization of the k matrices B_i; note that the current implementation uses sequential Cholesky solvers, but parallel Cholesky solvers could be used for increasing the degree of parallelism of the approach.
- Computation of $\beta^0 = \bar{b}^0 - \sum_{i=1}^k C_i^T B_i^{-1} \bar{b}^i$, which requires k backsolves on the factorizations of B_i and matrix-vector products of the form $C_i^T v^i$.
- For each iteration of the PCG, computation of $(D - \sum_{i=1}^k C_i^T B_i^{-1} C_i)v$, which requires backsolves on the factorizations of B_i and matrix-vector products of the form $C_i v^i$ and $C_i^T v^i$.
- Computation of $\beta^i = \bar{b}^i - C_i \Delta y^0$, which requires matrix-vector products of the form $C_i v^i$.
- Solution of the systems $B_i \Delta y^i = \beta^i$.

Hence, most of the parallelization effort boils down to performing in parallel the factorization of the B_is, backward and forward substitution with these factorizations and matrix-vector products involving C_i or C_i^T. Thus, there is no need for sophisticated implementations of parallel linear algebra routines. Note that higher-order preconditioners ($h > 0$) would complicate somehow the above scheme, but the basic blocks would remain the same.

Although the above procedures are by far the most important, a number of other minor steps can be easily parallelized, such as the computation of the other primal and dual directions (Δx^i, Δz^i, Δw^i), the computation of the primal and dual steplenghts α_P and α_D, the updating of the current primal and dual solution, the computation of the primal and dual objective function values and so on. It is easy to see that all the data concerning one given commodity i (x^i, c^i, u^i, y^i, $w^i \ldots$) can be stored in the local memory of the one processor that is in charge of that commodity, and it is never required by other processors. This ensures a good "locality" of data, and a low need for inter-processor communication. It should also be noted that the number of operations required for each commodity is the same, which guarantees the load balancing between processors, at least as long as the number of commodities assigned to each processor is the same.

4.1 Parallel Programming Environment

The parallel version of the IPM code, pIPM, has been developed on the Silicon Graphics Origin2000 (SGI O2000) server located at the European Center for Parallelism of Barcelona (CEPBA), running an IRIX64 6.5 Unix operating system. Like most of the current parallel architectures, the SGI O2000 offers both message-passing and shared-memory programming paradigms, although the main memory is physically distributed among the processors. The server has

64 MIPS R10000 processors running at 250Mhz, each of them with 32+32Kb L1 cache and 4Mb L2 cache and credited of 14.7 SPECint95 and 24.5 SPECfp95. A total of 8Gb of memory is distributed among these processing elements. This computer appeared at position 275 of the TOP500 November 1998 supercomputer sites list [10].

The default programming style supported by the SGI O2000 is a custom shared-memory version of C [24], with parallel constructs specified by means of compiler directives (#pragmas). Placement of the memory on the processors and communication is hidden to the programmer and automatically performed by the system. The main advantage of this choice is ease of portability: existing codes can be parallelized with a limited effort. It is even possible to avoiding maintaining two different versions (sequential and parallel) of the same code, which is important to optimize the development efforts.

However, this programming style also has a number of drawbacks, mainly a limited control over memory ownership and limited support for vector-broadcast and vector-reduce operations. Placement of the data structures in the local memory of the processors can be only partly (and indirectly) influenced by the programmer. Also, the granularity of memory placement is that of the virtual memory pages (16K) rather than that of the individual data structures. All this can result in cache misses and page faults from the local memory of each processor, decreasing the performance of the parallel codes. Although advanced directives allow a more detailed control over these features, the use of those directives requires a more extensive rewriting of the code, thus loosing part of the benefits in terms of portability and ease of maintenance. Because of that, the computational results presented in Section 5 were obtained with the default data distribution provided by the system (the same used in [2]). However, the assignment of commodities to processors was optimized for this distribution, hopefully limiting the possible negative effects. The limited support for broadcast/reduce operations is understandable in a shared-memory oriented language; however, it may result in poorer performances for codes, like pIPM, where these operations amount at almost the totality of the communication time.

5   Computational Results

5.1   The Instances

Three sets of multicommodity instances were used for the computational experiments. The first is made up of 18 problems obtained with an improved version of Ali and Kennington's Mnetgen generator [11]. These instances are very large (up to about 2.5 millions of variables and 260,000 constraints), with the number of commodities which varies from very few (8) to quite many (512). This is useful for characterizing the trends in the performances of the code as the number of commodities varies [7, 11].

The second set consists of ten of the PDS (Patient Distribution System) problems. These problems arise from a logistic model for evacuating patients

8   J. Castro and A. Frangioni

from a place of military conflict. The different instances arise from the same basic scenario by varying the time horizon, i.e., the number of days covered by the model. The PDS problems has been considered, until recently, essentially impossible to solve with a high degree of accuracy. Although this has changed, they are still quite challenging multicommodity instances.

The third set of problems is made of the four Tripart problems and of the Gridgen1 problem. These instances were obtained with the Tripartite generator and with a variation for multicommodity flows of the Gridgen generator [4]. These are very difficult multicommodity flow instances, as shown in Section 5.3.

The dimensions of each problem are reported in Tables 1, 2 and 3. Columns "m", "n", and "k" show the number of nodes, arcs, and commodities. Columns "$\bar{n}$" and "$\bar{m}$" give the number of variables and constraints of the linear problem. All the instances can be downloaded from

<p align="center">http://www.di.unipi.it/di/groups/optimize/Data.</p>

5.2   Performance Measures

The following well-known performance measures [3] will be considered for assessing the performances of pIPM. Denoting by T_p the execution time obtained with p processors, the *speedup* S_p with p processors can be defined as $S_p = T_1/T_p$. The fraction of the sequential execution time consumed in the parallel region of the code will be denoted by f; values of f close to 1 are necessary in order to obtain good speedups, as demonstrated by *Amdahl's law*

$$S_p \leq \overline{S_p} = \frac{1}{f/p + (1 - f)} \leq \frac{1}{(1 - f)} .$$

The *efficiency* with p processors is

$$E_p = \frac{S_p}{p} \leq \overline{E_p} = \frac{\overline{S_p}}{p} .$$

E_p represents the fraction of the time that a particular processor (of the p available) is usefully employed during the execution of the algorithm. $\overline{S_p}$ and $\overline{E_p}$ are respectively the *ideal* speedup and efficiency, the maximum ones that can be obtained due to the inherent serial bottlenecks in the algorithm.

Another interesting performance measure is the *absolute* speedup, obtained by replacing T_1 with the execution time of the *best* serial algorithm known. This is usually difficult to obtain, and it will be discussed separately.

5.3   The Results

Tables 1, 2 and 3 show the computational results obtained. Columns "IP" and "PCG" report the total number of interior-point and PCG iterations, respectively. Column "f" gives the fraction of the total sequential time consumed in the parallel region of the code. Column "p" gives the number of processors used

Table 1. Dimensions and results for the Mnetgen problems.

	m	n	k	$\bar{n}$	$\bar{m}$	f	IP	PCG	p	T_p	S_p	$\bar{S}_p$	E_p	$\bar{E}_p$
128-8	128	1089	8	9801	2113	92.2	42	831	1	3.2	1.0	1.0	1.0	1.0
									8	2.1	1.5	5.2	0.2	0.6
128-16	128	1114	16	18938	3162	95.1	48	2530	1	14.3	1.0	1.0	1.0	1.0
									8	7.7	1.9	6.0	0.2	0.7
									16	8.0	1.8	9.2	0.1	0.6
128-32	128	1141	32	37653	5237	95.4	56	2355	1	32.1	1.0	1.0	1.0	1.0
									8	12.9	2.5	6.1	0.3	0.8
									16	13.8	2.3	9.5	0.1	0.6
									32	19.6	1.6	13.2	0.1	0.4
128-64	128	1171	64	76115	9363	97.1	72	5480	1	139.2	1.0	1.0	1.0	1.0
									8	39.7	3.5	6.7	0.4	0.8
									16	34.7	4.0	11.1	0.3	0.7
									32	28.6	4.9	16.9	0.2	0.5
									64	40.3	3.5	22.6	0.1	0.4
128-128	128	1204	128	155316	17588	96.6	85	5033	1	409.2	1.0	1.0	1.0	1.0
									8	74.4	5.5	6.5	0.7	0.8
									16	122.8	3.3	10.6	0.2	0.7
									32	122.7	3.3	15.6	0.1	0.5
									64	73.3	5.6	20.4	0.1	0.3
256-8	256	2165	8	19485	4213	95.6	57	2713	1	20.7	1.0	1.0	1.0	1.0
									8	8.3	2.5	6.1	0.3	0.8
256-16	256	2308	16	39236	6404	96.5	59	3465	1	58.0	1.0	1.0	1.0	1.0
									8	21.0	2.8	6.4	0.3	0.8
									16	21.3	2.7	10.5	0.2	0.7
256-32	256	2314	32	76362	10506	97.3	67	5438	1	252.2	1.0	1.0	1.0	1.0
									8	52.6	4.8	6.7	0.6	0.8
									16	44.2	5.7	11.4	0.4	0.7
									32	54.6	4.6	17.4	0.1	0.5
256-64	256	2320	64	150800	18704	98.0	80	7644	1	757.3	1.0	1.0	1.0	1.0
									8	128.5	5.9	7.0	0.7	0.9
									16	93.7	8.1	12.3	0.5	0.8
									32	99.1	7.6	19.8	0.2	0.6
									64	169.3	4.5	28.3	0.1	0.4
256-128	256	2358	128	304182	35126	98.8	98	12535	1	2672.1	1.0	1.0	1.0	1.0
									8	351.3	7.6	7.4	1.0	0.9
									16	298.7	8.9	13.6	0.6	0.8
									32	257.0	10.4	23.3	0.3	0.7
									64	263.5	10.1	36.4	0.2	0.6
256-256	256	2204	256	566428	67740	98.9	107	16901	1	6725.1	1.0	1.0	1.0	1.0
									8	1219.7	5.5	7.4	0.7	0.9
									16	763.4	8.8	13.7	0.6	0.9
									32	502.0	13.4	23.9	0.4	0.7
									64	477.9	14.1	37.8	0.2	0.6
512-8	512	4373	8	39357	8469	96.4	66	3870	1	90.5	1.0	1.0	1.0	1.0
									8	22.9	4.0	6.4	0.5	0.8
512-16	512	4620	16	78540	12812	97.6	73	5364	1	322.3	1.0	1.0	1.0	1.0
									8	72.0	4.5	6.8	0.6	0.9
									16	63.1	5.1	11.8	0.3	0.7
512-32	512	4646	32	153318	21030	98.8	103	22460	1	2721.4	1.0	1.0	1.0	1.0
									8	454.7	6.0	7.4	0.7	0.9
									16	299.3	9.1	13.6	0.6	0.8
									32	289.3	9.4	23.3	0.3	0.7
512-64	512	4768	64	309920	37536	99.2	95	27004	1	9244.5	1.0	1.0	1.0	1.0
									8	1271.5	7.3	7.6	0.9	0.9
									16	702.8	13.2	14.3	0.8	0.9
									32	507.9	18.2	25.6	0.6	0.8
									64	563.8	16.4	42.6	0.3	0.7
512-128	512	4786	128	617394	70322	99.3	112	28631	1	19385.9	1.0	1.0	1.0	1.0
									8	3237.0	6.0	7.6	0.7	1.0
									16	1780.6	10.9	14.5	0.7	0.9
									32	1271.5	15.2	26.3	0.5	0.8
									64	848.5	22.8	44.4	0.4	0.7
512-256	512	4810	256	1236170	135882	99.5	130	32676	1	43251.2	1.0	1.0	1.0	1.0
									8	7401.6	5.8	7.7	0.7	1.0
									16	5306.7	8.2	14.9	0.5	0.9
									32	2783.7	15.5	27.7	0.5	0.9
									64	2205.9	19.6	48.7	0.3	0.8
512-512	512	4786	512	2455218	266930	99.6	194	48229	1	135753.7	1.0	1.0	1.0	1.0
									8	25257.7	5.4	7.8	0.7	1.0
									16	14198.4	9.6	15.1	0.6	0.9
									32	8325.3	16.3	28.5	0.5	0.9
									64	5226.0	26.0	51.1	0.4	0.8

Table 2. Dimensions and results for the PDS problems.

	m	n	k	$\bar{n}$	$\bar{m}$	f	IP	PCG	p	T_p	S_p	$\bar{S}_p$	E_p	$\bar{E}_p$
PDS1	126	372	11	4464	1758	83.3	30	169	1	0.7	1.0	1.0	1.0	1.0
									6	0.5	1.3	3.3	0.2	0.5
									11	0.7	0.9	4.1	0.1	0.4
PDS10	1399	4792	11	57504	20181	94.7	66	1107	1	44.8	1.0	1.0	1.0	1.0
									6	25.3	1.8	4.7	0.3	0.8
									11	24.6	1.8	7.2	0.2	0.7
PDS20	2857	10858	11	130296	42285	96.6	69	1911	1	254.1	1.0	1.0	1.0	1.0
									6	70.9	3.6	5.1	0.6	0.9
									11	62.6	4.1	8.2	0.4	0.7
PDS30	4223	16148	11	193776	62601	97.9	92	3835	1	777.1	1.0	1.0	1.0	1.0
									6	206.4	3.8	5.4	0.6	0.9
									11	189.2	4.1	9.1	0.4	0.8
PDS40	5652	22059	11	264708	84231	97.9	73	1872	1	1288.1	1.0	1.0	1.0	1.0
									6	258.4	5.0	5.4	0.8	0.9
									11	194.1	6.6	9.1	0.6	0.8
PDS50	7031	27668	11	332016	105009	98.8	100	4711	1	3486.4	1.0	1.0	1.0	1.0
									6	727.3	4.8	5.7	0.8	0.9
									11	530.1	6.6	9.8	0.6	0.9
PDS60	8423	33388	11	400656	126041	99.0	106	5215	1	6262.0	1.0	1.0	1.0	1.0
									6	1252.4	5.0	5.7	0.8	1.0
									11	745.4	8.4	10.0	0.8	0.9
PDS70	9750	38396	11	460752	145646	99.2	116	7015	1	10873.8	1.0	1.0	1.0	1.0
									6	2112.2	5.1	5.8	0.9	1.0
									11	1268.5	8.6	10.2	0.8	0.9
PDS80	10989	42472	11	509664	163351	99.2	107	3768	1	8855.0	1.0	1.0	1.0	1.0
									6	1726.3	5.1	5.8	0.9	1.0
									11	1093.8	8.1	10.2	0.7	0.9
PDS90	12186	46161	11	553932	180207	99.4	135	9357	1	20784.3	1.0	1.0	1.0	1.0
									6	3950.5	5.3	5.8	0.9	1.0
									11	2447.8	8.5	10.4	0.8	0.9

Table 3. Dimensions and results for the Tripart and Gridgen problems.

	m	n	k	$\bar{n}$	$\bar{m}$	f	IP	PCG	p	T_p	S_p	$\bar{S}_p$	E_p	$\bar{E}_p$
Tripart1	192	2096	16	35632	5168	93.6	65	3733	1	34.9	1.0	1.0	1.0	1.0
									4	21.3	1.6	3.4	0.4	0.8
									8	17.9	1.9	5.5	0.2	0.7
									16	19.6	1.8	8.2	0.1	0.5
Tripart2	768	8432	16	143344	20720	91.8	63	2652	1	156.6	1.0	1.0	1.0	1.0
									4	71.6	2.2	3.2	0.5	0.8
									8	55.4	2.8	5.1	0.4	0.6
									16	60.3	2.6	7.2	0.2	0.4
Tripart3	1200	16380	20	343980	40380	94.9	84	9343	1	1140.7	1.0	1.0	1.0	1.0
									4	408.4	2.8	3.5	0.7	0.9
									10	300.5	3.8	6.9	0.4	0.7
									20	304.8	3.7	10.2	0.2	0.5
Tripart4	1050	24815	35	893340	61565	95.6	96	8498	1	3273.2	1.0	1.0	1.0	1.0
									5	893.7	3.7	4.3	0.7	0.9
									7	721.5	4.5	5.5	0.6	0.8
									35	601.1	5.4	14.0	0.2	0.4
Gridgen1	1025	3072	320	986112	331072	99.5	173	49981	1	37234.9	1.0	1.0	1.0	1.0
									8	10533.2	3.5	7.7	0.4	1.0
									16	7678.7	4.8	14.9	0.3	0.9
									32	4426.5	8.4	27.7	0.3	0.9
									64	3248.6	11.5	48.7	0.2	0.8

in the execution. "T_p" denotes the execution (wall-clock) time, excluding initializations. Columns "S_p" and "E_p" give respectively the observed speedups and efficiencies, while columns "$\overline{S_p}$" and "$\overline{E_p}$" report their ideal values.

Analyzing the results, the following trends emerge:

- f is always fairly large, and increases with the problem size; the largest problems attain very high ideal efficiencies. This indicates that the approach has a good potential for scalablility, at least in theory, for very large scale problems.
- For fixed p and k, E_p almost always increases with the size of the underlying network, in all three groups of instances. This is reasonable: the computational burden of the PCG iteration grows quadratically with the number of nodes, while the communication cost grows only linearly. This seems to indicate that the approach is especially suited for problems where the size of the network is large w.r.t. the number of commodities. Remarkably, IPM has been shown to be particularly efficient, at least w.r.t. decomposition approaches, exactly for this kind of instances [11].
- Keeping p and the size of the network fixed, E_p initially increases with k; however for "large" values of k E_p stalls, and may even decrease. This phenomenon, clearly visible in the Mnetgen results, is difficult to explain. For fixed p, increasing k can, in theory, only increase the fraction of time that is spent in the parallel part of the algorithm, while the sequential bottleneck and the communication requirements should remain the same. Indeed, $\overline{E_p}$ is monotonically nondecreasing with k. This decrease in efficiency is most likely an effect of the page-based memory placement, which may cause data logically pertaining to one processor to be phisically located on another.
- For any fixed instance, E_p obviously decreases as p increase; unfortunately, the decrease is much faster than that predicted by $\overline{E_p}$, so that the gap between E_p and $\overline{E_p}$ increases with p. However, for fixed p the gap decreases when the size of the network increase, and a similar—although less clear— trend seems to exist w.r.t. k. Thus, whatever mechanism be responsible for this discrepancy between E_p and $\overline{E_p}$, its effects seem to lessen as the instances grow larger.

Since, except for PDS problems with $p = 6$, each processor is assigned the same number of commodities, there can be no load imbalance between the processors. Thus, the gap between E_p and $\overline{E_p}$ can only be explained as being due to communication time. Indeed, pIPM requires more communication than most other parallel codes for multicommodity flows. Most of communication occurs during the computation of $\left(D - \sum_{i=1}^{k} C_i^T B_i^{-1} C_i \right) v$, where v is the current estimate of the solution of (8), at each PCG iteration. This requires first the broadcast of v from the "master" processor (the one executing the serial-only part of the code) to all the other processors, followed by a vector-reduce operation to accumulate all the partial results $C_i^T B_i^{-1} v$ back to the "master" processor. The amount of communication is essentially the same as in the decomposition approaches [7, 12, 21], and substantially lower than that of the other specialized

12   J. Castro and A. Frangioni

parallel interior-point codes [16, 9], which need to share the (dense) matrices $C_i^T B_i^{-1} C_i$ in order to form the Schur complement H. However, in pIPM communication occurs at *every PCG iteration*, i.e., much more often than in decomposition codes. The other specialized parallel interior-point codes have a much smaller number of communication "rounds", one for each interior-point iteration, although each round is more expensive.

Thus, pIPM may be inherently more vulnerable to slowdowns induced by communication costs. Indeed, the efficiency of pIPM seems to be, on average, somehow worse than that of the approach in [16], even though direct comparison is difficult due to the different sets of test problems. The instances used in [16] are much smaller, and the cost of forming and factorizing H grows rapidly with the size of the problem.

Furthermore, the current implementation of pIPM, using the parallel constructs available in the SGI O2000 C compiler [24], is not aggressively optimized particularly in the two critical operations, i.e., broadcasts and vector-reduces. Both are currently obtained by means of read/write operations to shared vectors, which are presumably less efficient than the typical system-provided implementation which exploits information about the topology of the interconnection network and the available communication hardware. Also, a part of the communication overhead could be due to a non-optimal placement of the data structures in the local memory of the processors, especially at the boundaries of the virtual memory pages. Thus, we believe that there is still room for (potentially large) reductions of the gap between the observed and the theoretical speedup/efficiency of the code. However, pIPM already attains quite satisfactory efficiencies in some instances, most notably the largest PDS problems.

Table 4. Comparing Cplex 6.5 and IPM on the Tripart and Gridgen problems.

Problem	IPM	Cplex 6.5
Tripart1	40	74
Tripart2	249	627
Tripart3	1584	2851
Tripart4	4983	33235
Gridgen1	126008	$\geq$ 2.8e+6

As far as the absolute speedup is concerned, IPM is known not to be the fastest sequential code for some of the test instances. In [11], a bundle-based decomposition approach has been shown to outperform IPM on the Mnetgen instances, while IPM was competitive on the PDS problems. Furthermore, recent developments in the field of simplex methods [18] have lead to impressive performance improvements for these algorithms on multicommodity flow problems. Nowadays, even the largest PDS problems can be solved in less than an hour of CPU with the state-of-the-art simplex code Cplex 6.5. However, the simplex method is not easily parallelized. Furthermore, other multicommodity problems, like the Tripart and the Gridgen, are much more difficult to solve; ϵ-approximation algorithms can approximatively solve them in a relatively short time [4], but only if the required accuracy is not high. On these instances, the

interior-point algorithm in Cplex 6.5 is far more efficient than the dual simplex, but it is in turn largely outperformed by IPM, as shown in Table 4. Columns "IPM" and "Cplex 6.5" represents the running time required for the solution of the problem by IPM and Cplex 6.5, respectively, on a Sun Ultra2 2200/200 workstation (credited of 7.8 SPECint95 and 14.7 SPECfp95) with 1Gb of main memory. Thus, for the largest and more difficult instances of the set, pIPM provides a competitive approach.

6 Conclusions and Future Research

The parallel code pIPM presented in this work can be an efficient tool for the solution of certain types of large and difficult multicommodity problems. Quite good speedups are achieved in some instances, such as the large PDS problems. In other cases, a gap between the ideal efficiency and the observed one exists. However, we are confident that a more efficient implementation of reduce/broadcast operations and a better placement of data structures—which could mean using MPI or PVM as parallel environments—can make pIPM even more competitive on a widest range of multicommodity instances.

References

1. Adler, I., Resende, M.G.C., Veiga, G.: An implementation of Karmarkar's algorithm for linear programming. Math. Prog. **44** (1989) 297–335
2. Andersen, E.D., Andersen, K.D.: A parallel interior-point algorithm for linear programming on a shared memory machine. CORE Discussion Paper **9808** (1998), CORE, Louvain-La-Neuve, Belgium.
3. Bertsekas, D.P., Tsitsiklis, J.N.: Parallel and Distributed Computation. Prentice-Hall, Englewood Cliffs (1995).
4. Bienstock D.: Approximately solving large-scale linear programs. I: Strengthening lower bounds and accelerating convergence. CORC Report **1999–1** (1999), Columbia University, NY.
5. Castro, J.: A specialized interior-point algorithm for multicommodity network flows. SIAM J. on Opt. (to appear).
6. Castro, J.: Computational experience with a parallel implementation of an interior-point algorithm for multicommodity network flows. M. Powell and S. Scholtes (eds.), Proceedings of the 19th IFIP TC7 Conference, Kluwer. (to appear)
7. Cappanera, P., Frangioni, A.: Symmetric and asymmetric parallelization of a cost-decomposition algorithm for multi-commodity flow problems. Technical Report **TR-96-36** (1996), Dip. di Informatica, Università di Pisa, Italy.
8. Coleman, T.F., Czyzyk, J., Sun, C., Wagner, M., Wright, S.J.: pPCx: parallel software for linear programming. Proceedings of the Eight SIAM Conference on Parallel Processing in Scientific Computing, SIAM, March 1997.
9. De Silva, A., Abramson, D.A.: A parallel interior-point method and its application to facility location problems. Computational Optimization and Applications **9**(3) (1998) 249–273.
10. Dongarra, J.J., Meuer, H.W., Strohmaier, E.: TOP500 supercomputer sites. Technical Report **UT-CS-98-404** (1998), Computer Science Dept. University of Tennessee.

14   J. Castro and A. Frangioni

11. Frangioni, A., Gallo, G.: A bundle type dual-ascent approach to linear multicommodity min cost flow problems. INFORMS J. on Comp. **11**(4) (1999) 370–393.
12. Gondzio, J., Sarkissian, R., Vial, J.-P.: Parallel implementation of a central decomposition method for solving large scale planning problems. HEC Technical Report **98.1** (1998).
13. Jessup, E.R., Yang, D., Zenio, S.A.: Parallel factorization of structured matrices arising in stochastic programming. SIAM J. on Opt. **4**(4) (1994) 833–846.
14. Kamath, A.P., Karmarkar, N.K., Ramakrishnan, K.G.: Computational and complexity results for an interior-point algorithm on multicommodity flow problems. Technical Report **TR-21-93** (1993), Dip. di Informatica, Università di Pisa, Italy.
15. Kontogiorgis, S., De Leone, R., Meyer, R.R.: Alternating directions splitting for block angular parallel optimization. JOTA **90**(1) (1996) 1–29.
16. Lustig, I.J., Li, G.: An implementation of a parallel primal-dual interior-point method for block-structured linear programs. Computational Optimization and Applications **1** (1992) 141–161.
17. Lustig, I.J., Rothberg, E.: Gigaflops in linear programming. O.R. Letters **18**(4) (1996) 157–165.
18. McBride, R.D.: Advances in Solving the Multicommodity Flow Problem. SIAM J. on Opt. **8**(4) (1998) 947–955.
19. Medhi, D.: Parallel bundle-based decomposition for large-scale structured mathematical programming problems. Annals of O.R. **22** (1990) 101–127.
20. Ng, E., Peyton, B.W.: Block sparse Cholesky algorithms on advanced uniprocessor computers. SIAM J. Sci. Comput. **14** (1993) 1034–1056.
21. Pinar, M.C, Zenios, S.A.: Parallel decomposition of multicommodity network flows using a linear-quadratic penalty algorithm. ORSA J. on Comp. **4** (1992) 235–249.
22. Portugal, L.. Resende, M.G.C.. Veiga, G.. Júdice, J.: A truncated interior-point method for the solution of minimum cost flow problems on an undirected multicommodity flow network. Proceedings of First Portuguese National Telecommunications Conference, Aveiro, Portugal (1997) 381–384 (in Portuguese).
23. Rosen, J.B. (ed.): Supercomputers and large-scale optimization: algorithms, software, applications. Annals of O.R. **22** (1990).
24. Silicon Graphics Inc.: C Language Reference Manual (1998).
25. Schultz, G, Meyer, R.: An interior-point method for block-angular optimization. SIAM J. on Opt. **1** (1991) 583–682.
26. Wright, S.J.: Primal-Dual Interior-Point Methods. SIAM, Philadelphia, PA (1997).

A Parallel Algorithm for the Simulation of the Dynamic Behaviour of Liquid-Liquid Agitated Columns

E. F. Gomes [§], L. M. Ribeiro[¶], P. F. R. Regueiras[¶] and J. J. C. Cruz-Pinto[*]

§ Instituto Superior de Engenharia do Porto, Porto, Portugal, ¶ Faculdade de Engenharia da Universidade do Porto, Porto, Portugal, *Universidade de Aveiro, Aveiro, Portugal

`lmr@fe.up.pt`

Abstract. Simulation of the dynamic behaviour of liquid-liquid systems is of prominent importance in many industrial fields. Algorithms for fast and reliable simulation of single stirred vessels and extraction columns have already been published by some of the present authors. In this work, we propose a methodology to develop a parallel version of a previously validated sequential algorithm, for the simulation of a liquid-liquid Kühni column. We also discuss the algorithm implementation in a distributed memory parallel-computing environment, using MPI. Despite the difficulties encountered to preserve efficiency in the case of a heterogeneous cluster, the results demonstrate performance improvements that clearly indicate that the approach followed may be successfully extended to allow real-time plant control applications.

Key words: Distributed Memory Parallel Systems; MPI; Simulation of Liquid-Liquid Systems.

1. Introduction

The mass transfer efficiency of liquid-liquid agitated systems is highly dependent on the hydrodynamics of the dispersed phase, namely of the drop break-up and coalescence frequencies that result from the turbulence induced by agitation. In reacting systems, this behaviour is also of fundamental importance to the overall rate and selectivity of the process. A comprehensive and synthetic discussion about the behaviour of liquid-liquid systems is found in Ramkrishna's work [1].

Knowledge of the dynamic behaviour of liquid-liquid systems is still limited, in particular when it comes to its implementation as physically accurate, fast and reliable algorithms, with effective predictive power and suitable for real-time plant control applications [2]. Potential fields of practical use of this knowledge base encompass very broad segments of chemical technology, including the recovery of important non-renewable resources or the removal of dangerous substances.

Ribeiro L. M. [3] and Ribeiro L. M. *et al.* [4] published innovative algorithms for directly (numerically) solving the population balance equation for the simulation of

---

Candidate to the Best Student Paper Award

the full trivariate (drop volume, v, solute concentration, c, and age, •) unsteady-state behaviour of interacting liquid-liquid dispersions, in single continuous (or batch) stirred vessels. Not only the start-up period towards the steady-state was simulated but also the system's response to disturbances in the main operating variables (mean residence time, dispersed phase hold-up, agitation power input density, feed drop volume distribution and dispersed and continuous phase solute concentrations). The methodology used was later applied to a simplified version of the algorithm, that calculates the drop size distribution and the mean and standard deviation of solute concentration within each volume class [5]. This methodology was further extended to simulate the behaviour of a liquid-liquid extraction column [6].

The aim of this paper is to show that, using low cost high performance computing environments and the above referred methodology, it is possible to simulate in detail the dynamics of stirred liquid-liquid extraction columns, with execution times suitable for prediction of the behaviour of these systems and for control purposes.

2. The sequential algorithm

Following the experimental work carried out by Gomes [7] in a Kühni pilot plant column of the Technical University of Munich, a sequential algorithm was developed to trace its dynamics [6]. This column has 150mm of internal diameter and 36 stages, each 70 mm high.

A Kühni column may be adequately described as a sequence of agitated vessels with back mixing and forward mixing effects on the movement of the dispersed phase along the column. The hydrodynamic phenomena of break-up and coalescence of the individual drops of the dispersed phase was modeled using the population balance formulation of Coulaloglou and Tavlarides [8].

Besides the interaction phenomena, the transport of the drops from one stage to the next must also be modeled. The transport model used was based on the one described by Cruz-Pinto [9], taking into account the constriction factor calculated by Goldman [10] and the dispersion equation developed by Regueiras [11]. The mathematical model equations used are presented elsewhere [11].

From the mathematical model, the drop birth and death rates due to break-up, coalescence and drop movement along the column are calculated. Representing by $B(\overline{n}, t)$ and $D(\overline{n}, t)$ these source and sink terms, at time t and location $[\overline{n}, \overline{n} + d\overline{n}]$ of the drop phase space, the dynamics of the drop number density function $X(\overline{n}, t)$ is described by:

$$\frac{\partial}{\partial t} X(\overline{n}, t) + \frac{\partial}{\partial \overline{n}} \left[\frac{\partial \overline{n}}{\partial t} \cdot X(\overline{n}, t) \right] = B(\overline{n}, t) - D(\overline{n}, t) \tag{1}$$

To numerically solve the above population balance equation, a phase space-time discretization is used and drops are assumed to reside on cell sites. Drops move from cell to cell in the discretized phase-space at each time step. The numerical integration scheme involves the explicit calculation of time derivatives, with a first-order backward finite-difference method [4].

The sequential algorithm developed for the counter-current Kühni column simulation is able to predict the local drop size distributions and the local hold-up profiles of the column. The algorithm was implemented in C++ and the corresponding program is presently available for Windows 9x and Windows NT environments [6].

The program consists of two parts: the initialization of the system and the column simulation. The corresponding flowchart is presented in Fig. 1.

The main program starts reading all data needed to perform the simulation. This data includes the physical characteristics of the column, like the number of stages, stirrer diameter, height and diameter of each stage, the drop breakage, coalescence and transport model parameters, the physical properties of both phases, such as density, viscosity and interfacial tension, the operating conditions of the column, namely the flow rates of each phase and the stirrer rotational speed, the total simulated time, *tmax*, and the time interval, Δt, at which the program writes to a file the values of the column and system state variables.

At time $t=0$, the column variables are initialized to a standard initial state, corresponding to a column filled with continuous phase and no dispersed phase.

The program goes then into a loop where it writes the values of the column variables on a file, tests if the time reached the total simulated time value and, if not, calls the `TimeStep` routine to calculate the column status at time $t+\Delta t$. Then, it updates the value of t, and returns to the beginning of the loop. When *tmax* is reached the program exits the loop, writes global results to a file and terminates execution.

The routine `TimeStep` executes the simulation of the column for a period of time, Δt, between two consecutive `WriteData` calls. In order to accomplish this objective, the routine calls the `dXdt` routine for each column stage and, based on the death frequencies obtained, calculates a suitable step value for the integration. This value, *dt*, is then used to calculate the new values of the variables describing the state of the column. When the accumulated time reaches Δt, this routine is exited, returning control to the main program loop.

The routine `dXdt` calculates the drop birth and death frequencies inside a single column stage, as well as the number of drops per unit time exchanged with the contiguous stage. It also calculates the continuous phase flow rate between the same two stages. To perform these calculations, this routine needs the values of the statue variables at both stages. Only the auxiliary variables of the current stage are modified.
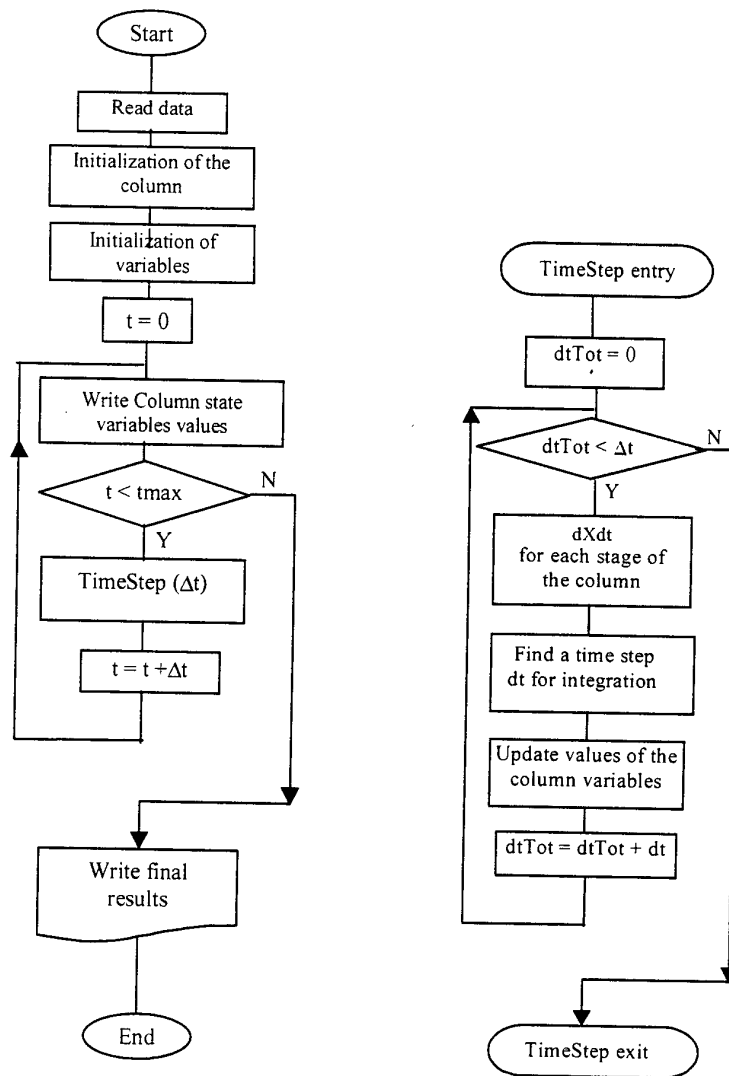
Fig. 1. Sequential algorithm and TimeStep routine

The hierarchy of the called routines and the routine tasks are outlined in Fig. 2 and Table 1, respectively.

The routine LLExtrColumns corresponds to the 'Initialization of the Column' box and to the 'Initialization of the variables' box. TimeStep and dXdt routines are designated on the flowchart for their own names.
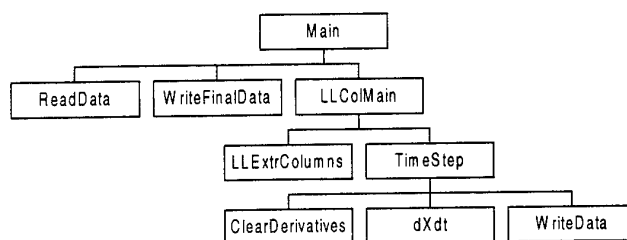

Fig. 2. The hierarchy of the called routines

ClearDerivatives	Prepares the variables for the calculations in dXdt.
dXdt	Calculates the drop birth and death frequencies of one stage.
LLColMain	Main part of the program; calls the routines.
LLExtrColmuns	Prepares each stage for the beginning of the simulation and calculates the inlet drop distributions.
TimeStep	Executes the simulation for a given period of time.
WriteData	Outputs to a file the results at the end of each time-step.
WriteFinalData	Outputs to a file the final results.

Table 1. Routine tasks

We have already shown that the results obtained with the sequential program for the hold-ups and the drop size distributions at different stages of the column are in good agreement with the experimental data, for several operating conditions of the column [7].

So far, the program doesn't include mass transfer calculations. With mass transfer, it is generally necessary to solve the population balance equation (1) in a tri-dimensional phase-space. In the present case, using a monovariate drop property (volume) distribution, the execution time achieved with a 120 MHz Pentium for one second of simulation time was four times longer than the real process, with a drop volume disctretization of 20 classes. Although already fast, in comparison to other resolution approaches [2], this algorithm needs to be further accelerated in order to be suitable for future control applications to liquid-liquid extraction columns, in mass transfer conditions. The introduction of excessive algorithm simplifications, other than those of the underlying mathematical model, are not desirable, as they would hide most of the information on the temporal behaviour of the dispersed phase properties distribution. This need to speedup the calculations was the motivation for

the development of a parallel version of the sequential program. This parallel version, implemented for a distributed memory parallel computing environment, is nowadays the only published promising approach to the future realistic simulation of various contactors, including extraction columns, and their control.

3. The parallelization approach

3.1 Initial considerations

A sequential C code was written for the algorithm to ensure that the calculations in each time step only need the results from the previous iteration.

The analysis of the logical units of this sequential code pointed out the methodology used to develop a parallel version of the algorithm. Table 2 clearly shows that the most time consuming routine is the one responsible for calculating the drop birth and death frequencies (due to drop breakage, coalescence, and transport) in each time step and for each column stage (dXdt routine). The time taken by the execution of the other routines is relatively insignificant and is not shown in Table 2. The parallel version of the algorithm is thus based on the partition of the calculation of these frequencies, for each time step, among the several processors available. The synchronization is made at the end of each iteration.

Name	Time (%)	Secs	Calls	Calls (ms/call)	Total (ms/call)
dXdt	87.40	2.29	5040	0.45	0.49
TimeStep	4.20	0.11	20	5.50	131.00

Table 2. The most time consuming routines

3.2 The MPI implementation

The parallel program was implemented in C for a distributed memory parallel-computing environment using MPI (MPICH, 1.1.2.).

The flowchart below shows that all of the processes call the TimeStep routine. In this routine, the master sends a sequence of stages for each one of the other processes, keeping the first group for itself. Each process also receives the last stage of the previous process, since this information is needed for the calculations. All the processes, including the master, contribute to the calculation, calling the dXdt routine. The master receives all the results sent by the other processes at the end of each time step and performs the control calculations, such as the overall hold-up and the verification of an eventual column flooding situation.

Master (process p0)   *Processes p1, ..., pn-1*
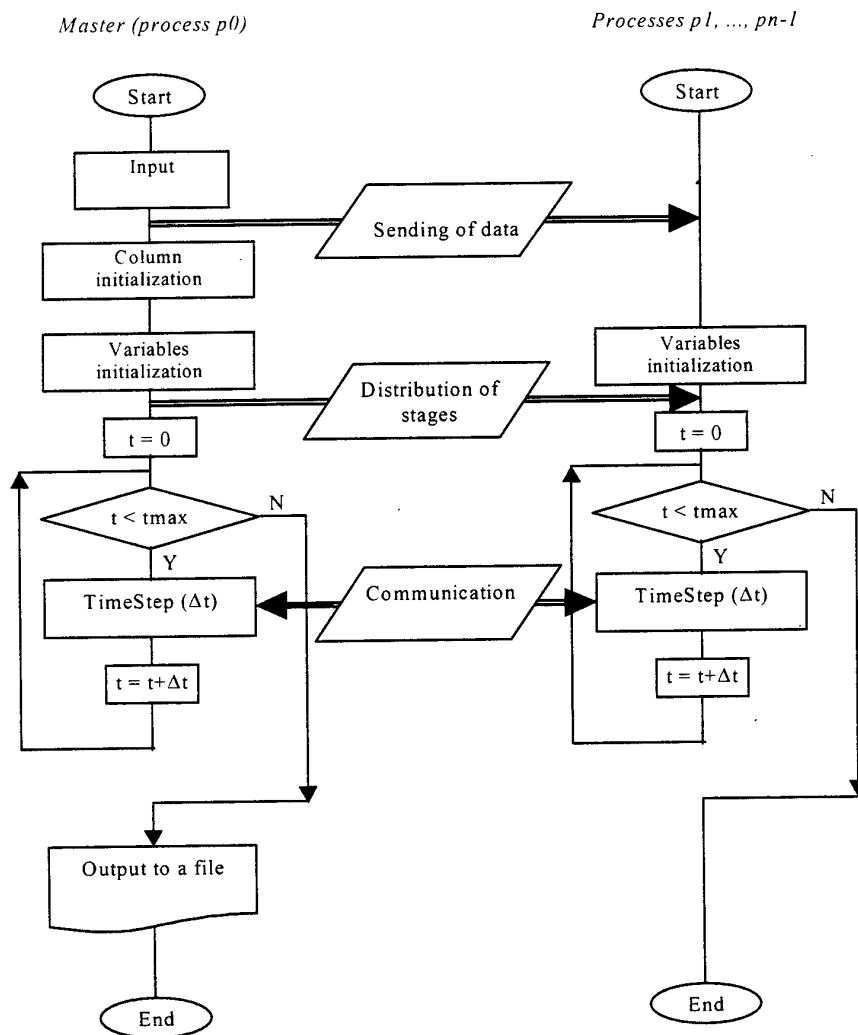
Fig. 3. The parallel algorithm
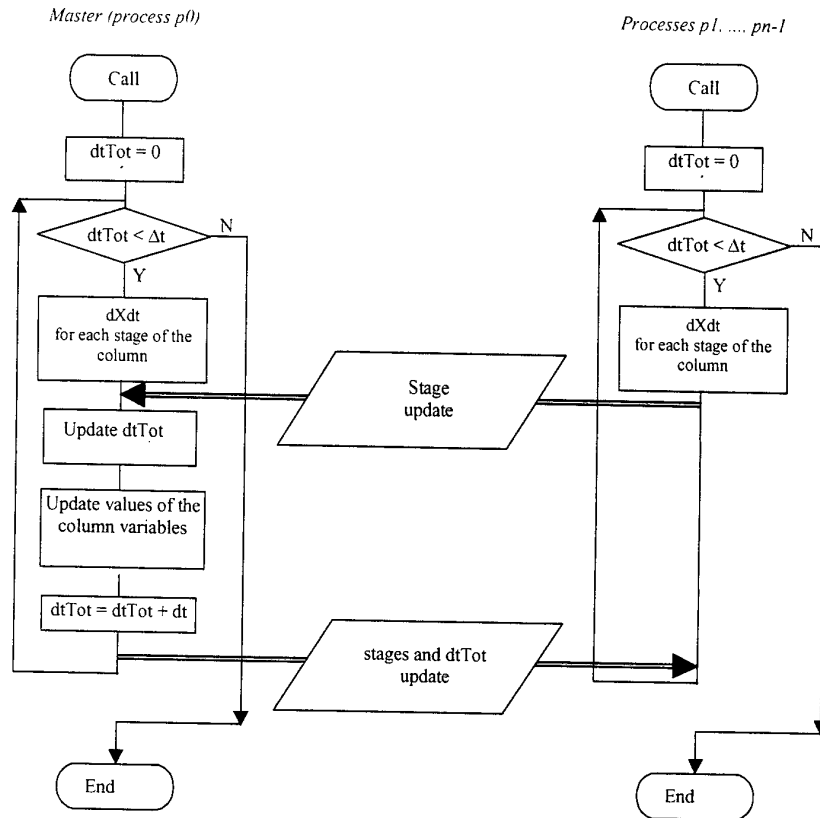
Fig. 4. The TimeStep routine

In order to minimize the overload due to information exchange, presently about 4KB for each stage (13,3 KB when mass transfer is included), every information was sent once (MPI_ISend), taking advantage of the *count* and *derived types* MPI parameters.

The program was first tested both on a heterogeneous cluster and on a homogeneous one. On the heterogeneous cluster, from the Engineering Faculty of the University of Porto, five Alpha processors were used, with different clock rates, 150 MHz (2 nodes), 175 MHz (2 nodes) and 266 MHz (1 node). A 100 Mbps FDDI crossbar switch (Digital Equipment Coorporation/Compaq GIGAswitch) connects these nodes. The operating system is True64 Unix v4.0E. On the homogeneous cluster, from the Dolphin [12] project of the Science Faculty of the University of Porto, four dual Pentium II, 300 MHz processors, interconnected by a Myrinet network, were used. The operating system was Linux Redhat 6.0.

Besides validation of the results, the possibility of using these different computation environments enabled us to identify problems in preserving efficiency for heterogeneous clusters [13]. The comparison of Fig.5 and Fig.6, that show the

monitor results of the *jumpshot* public domain utility, already discloses these problems. These figures show the inter-process communications for the heterogeneous cluster, with five processors, and for the homogeneous cluster, with six processors, both for a drop volume discretization of 20 classes. The black blocks represent the time consumed by the dXdt routine, and gray blocks refer to the TimeStep routine. The white arrows represent the stage exchanges between the processes.

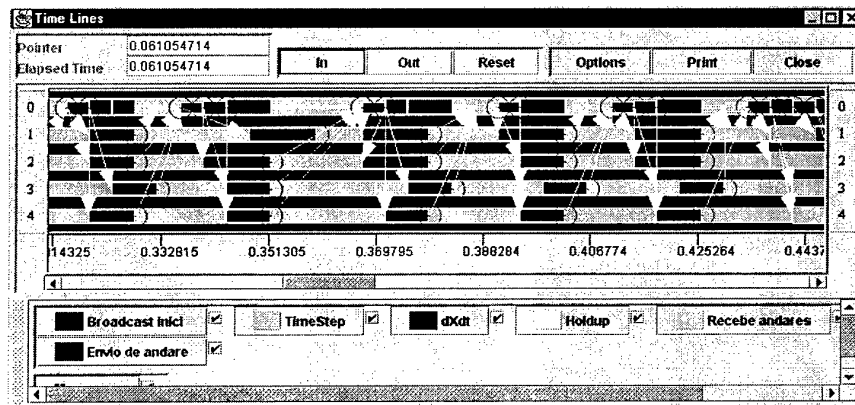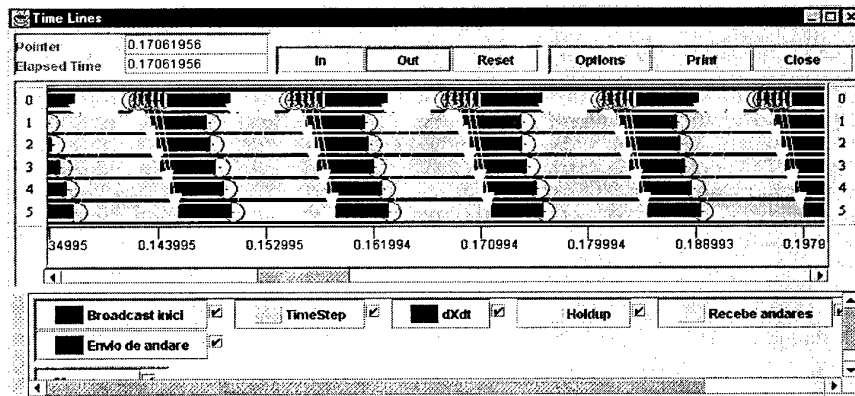**Fig. 5** *Jumpshot* result for the heterogeneous cluster

Fig. 6 *Jumpshot* result for the homogeneous cluster

On the homogeneous cluster, with a drop volume discretization of 100 classes, the results obtained with six processors showed speedups exceeding a factor of four (Fig.7). This result, for a realistic problem dimension, points out that parallelization pays off for the intended application [13].
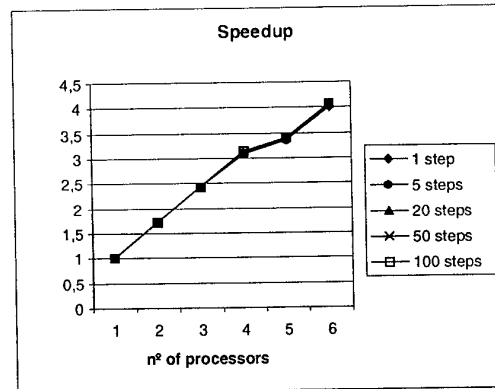
Fig. 7. Speedup for 100 classes, with the cluster of project Dolphin

4. Results and discussion

To envisage the future application of such parallel program in industry, a homogeneous dedicated cluster was selected. It is important to stress, at this point, that MPI doesn't respond dynamically to the potential inefficiencies caused by non-uniform computing speeds of the cluster nodes and the variability of shared resources [14].

The program was executed on the Beowulf Cluster of the Engineering Faculty of the University of Porto. The present configuration of this cluster of commodity PCs is one front-end node and twenty-two computing nodes. The front-end is a dual Pentium III 550 MHz processor, with 512 MB of memory and 18 GB of disk. Each computing node is a single 450 MHz Pentium III, with 128 MB of memory and 6 GB of disk. The nodes are connected using a Fast Ethernet BayNetworks 450-24 port switch. The operating system is Linux Slackware 7.0 [15]. The results obtained are presented in Fig. 8 and Fig. 9.

These results show speedups exceeding a factor of six, with eighteen processors, for a drop volume discretization of 100 classes. It can be observed that speedup, although increasing, shows some plateaus. For instance, between nine and eleven processors, speedup stabilizes and goes up again for twelve processors. Notice that nine and twelve divide thirty-six, which is the number of stages of the column. From twelve to seventeen processors we again have a plateau, and another at a higher level, from eighteen to twenty two processors. Eighteen also divides thirty-six. These performance leaps are related to the way in which we distribute the work for the various processors. First, when the number of processors divides the number of stages, the workload is equally distributed. Second, granularity decreases as communication time increases, and the calculation time per processor decreases.
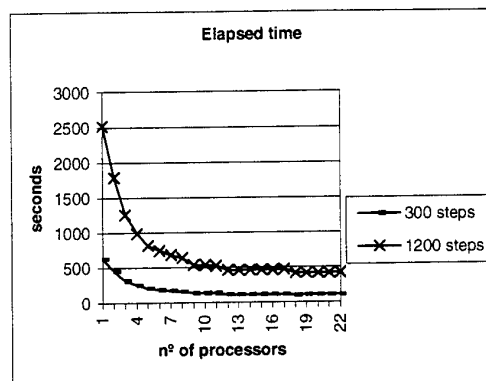
Fig. 8. Elapsed time for 100 classes

The speedup results for different discretizations of the drop phase-space, 50 and 100 drop volume classes, are shown in Fig. 9. For twenty-two processors and 300 time-steps, the results show a speedup increase from 3.71 to 5.97, being higher for the finer distribution. With 100 drop volume discretization classes and four processors, simulation is already faster than the real process.
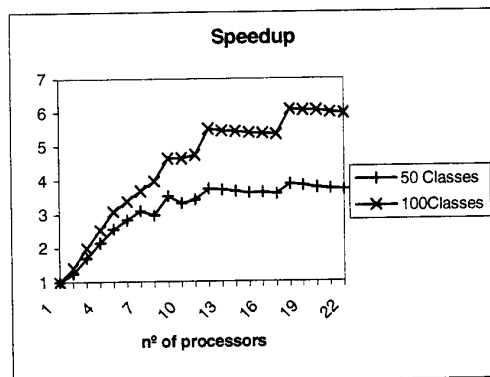
Fig. 9. Speedup for 50 and 100 classes

5. Conclusions and future work

The application that motivated this work was the simulation of the dynamic behaviour of liquid-liquid agitated columns. Execution times associated with sequential algorithms previously published by some of the authors need to be improved, in order to consider their application to real-time plant control applications.

Clustered systems, using commodity processors and standard Ethernet networks, are increasingly popular, in face of their low price/performance ratio.

We have shown that PC clusters are well suited for the intended application. The results presented in section 4 lead to the conclusion that parallelization pays off for the numerical technique used, based upon a space-time discretization and a stepping procedure, with explicit calculation of time derivatives. The fact that the speedup increases with the problem size is an important result for the future work, because mass transfer simulations involve much heavier calculations than the hydrodynamics.

Extensions of the algorithm to include mass transfer are presently under development, as well as studies concerning the optimization of the drop interaction constants and transport parameters.

On this version of the parallel program, the master is responsible for all global calculations, besides its own stage calculations, as a separate process. With this approach, all the communications are made only between the master and the other processes. Work is in progress to test another methodology, where all processes take part of the global calculations, implying communication between the i process and the $i-1$ process, instead of all process communications being with the master. This solution takes work from the master but increments communication between the processes. The analysis of the results will show whether, with this other communication and work distribution scheme, speedup can be further improved.

References

1. Ramkrishna D., *The Status of Population Balances*, Rev. Chem. Engng., **3**, 49, 1985.
2. Ribeiro L. M., Regueiras P. F. R., Guimarães M. M. L., Cruz-Pinto J. J. C., *Efficient Algorithms for the Dynamic Simulation of Agitated Liquid-Liquid Contactors*, Advances in Engineering Software, in print.
3. Ribeiro L. M., *Simulação Dinâmica de Sistemas Líquido-Líquido. Um novo Algoritmo com Potencialidades de Aplicação em Controlo*, Ph.D. Thesis, Universidade do Minho, Portugal, 1995.
4. Ribeiro L. M., Regueiras P. F. R., Guimarães M. M. L., Cruz-Pinto J. J. C., *The Dynamic Behavior of Liquid-Liquid Agitated Dispersions. II – Coupled Hydrodinamics and Mass Transfer*, Computers and Chem Engng., Vol. **21**, N°5, pp. 543-558, 1997.
5. Regueiras P. F. R., Ribeiro L. M., Guimarães M. M. L., Madureira C. M. N., Cruz-Pinto J. J. C., *Precise and Fast Computer Simulations of the Dynamic Mass Transfer Behaviour of Liquid-Liquid Agitated Contactors*, Value Adding Through Solvent Extraction, edited by D. C. Shallcross, R. Paimin, L. M. Prvcic, University of Melbourne, Vol. **1**, Proceedings of ISEC'96, pp. 1031-1036, 1996.
6. Regueiras P. F. R., Gomes M. L., Ribeiro L. M., Guimarães M. M. L., Cruz-Pinto J. J. C., *Efficient Computer Simulation of the Dynamics of an Agitated Liquid-Liquid Extraction Column*, Proceedings of the 7th International Chemical Engineering Conference, Lisboa, Portugal, September 1998.
7. Gomes M L. N., *Comportamento Hidrodinâmico de Colunas Agitadas Líquido-Líquido*, Ph.D. Thesis, Universidade do Minho, Portugal, 2000 (to be defended in April).
8. Coulaloglou C. A., Tavlarides L.L., *Description of Interation Processes in Agitated Liquid-Liquid Dispersions*, Chem. Engng Sci., **32**, 1289, 1977.
9. Cruz-Pinto J. J. C., *Experimental and Theorical Modelling Studies of the Hydrodynamic and Mass Transfer Processes in Countercurrent-Flow Liquid-Liquid Extraction Columns*, PhD. Thesis, The Victoria University, Manchester, 1979.

10. Goldman G., *Ermittlung und Interpretation von Kennlinienfelderen einer Gerührten Extraktionskolonne*, Ph.D. Thesis, Technical University of Munich, Germany, 1986.
11. Regueiras P. F. R., Ph.D. Thesis, in preparation.
12. http://www.ncc.up.pt/liacc/NCC/Projs/projectos_9.html [2000/03/15].
13. Gomes E. F., *Aplicação do Processamento Paralelo à Simulação Dinâmica das Colunas de Extracção Agitadas*, MSc. Thesis, Universidade do Porto, 1999.
14. Colajanni M., Cermele M., *DAME: An Environment for Preserving the Efficiency of Data-Parallel Computations on Distributed Systems*, IEEE Concurrency, Vol. 5, Nº 1, 1997
15. http://webforos.fe.up.pt:1313/perl/wwwboard/view_msg?id=10[2000/03/15].

Performance Analysis and Modeling of Regular Applications on Heterogeneous Workstation Networks

Andrea Clematis[1] and Angelo Corana[2]

[1] Istituto Matematica Applicata - Consiglio Nazionale Ricerche,
Via De Marini 6, 16149 Genova, Italy
E-mail: clematis@ima.ge.cnr.it
[2] Istituto Circuiti Elettronici - Consiglio Nazionale Ricerche,
Via De Marini 6, 16149 Genova, Italy
E-mail: corana@ice.ge.cnr.it

Abstract. Heterogeneous networks of workstations and/or personal computers (NOW) are increasingly used as a powerful platform for the execution of parallel applications.

Sometimes applications are developed having in mind this type of heterogeneous environment, but in most cases applications already developed for traditional parallel machines (homogeneous and dedicated) are ported to NOWs, resulting in performance degradation due in part to less efficient communications but more often to unbalancing.

In this work we propose a simple model able to analyze and predict performance on heterogeneous NOWs of regular data-parallel applications originally developed for ring or 2-D mesh topologies. To improve performance, the computation time on the various nodes must be as balanced as possible. This can be obtained in two ways: by heterogeneous data partitioning or by assigning to each node a number of processes proportionally to its relative power.

A test case based on matrix multiplication is analyzed and the results predicted by the model are compared with the ones collected experimentally.

Our analysis shows that an efficient porting of homogeneous data-parallel applications on heterogeneous NOWs is possible and can be achieved in most cases in a quite straightforward and effective way.

1 Introduction

In recent years networks of workstations and/or personal computers are increasingly used for the execution of parallel applications [7, 11]. Indeed technological advances make available nodes with high computing power and interconnecting networks with sufficiently high communication speed.

These systems constitute a viable alternative to classical parallel machines (which are homogeneous and dedicated) and have the advantages of a wide availability and a good price/performance ratio.

Main features of NOWs are: heterogeneity, since in most cases the various nodes are different, making a good balancing among nodes a critical aspect;

communication latency that is normally higher that the one in the 'true' parallel machines, imposing limits on fine grain computation.

A simple and effective way to achieve good efficiency on such platforms is the use of the master-worker programming model with the pool of tasks paradigm, which is self-balancing [10]. However, this approach is only feasible if tasks are independent. Moreover, it cannot be adopted if we are interested in the efficient and straightforward porting on NOWs of parallel applications which have been developed with different programming models for homogeneous and dedicated parallel systems.

Particularly, a number of data-parallel applications have been implemented on homogeneous systems with regular topologies such as ring and mesh using the SPMD model, obtaining loosely synchronous applications, well balanced and therefore providing a good efficiency.

If we execute applications belonging to this class on heterogeneous NOWs, the various nodes have in general different speeds, thus the fastest ones exibit a high idle time, resulting in a overall performance degradation. In order to minimize idle time, the computational work in each node must be as close as possible proportional to the computing power of the node.

Similar problems have been recently addressed by other authors. In [1] the problem arising with the use of grid algorithms on heterogeneous workstation networks is addressed, and solution based on sophisticated data allocation methods are proposed.

In this work we consider two possible strategies to obtain a good load balancing: a single process per node with heterogeneous data partitioning; homogeneous data partitioning assigning a different number of processes to each node, according to its computing power.

We propose a simple model able to evaluate performance in the various cases, taking into account the involved parameters at the application level (e.g. computational work and communication amount), at the architectural level (e.g. interconnection network speed) and at both levels (e.g. relative speed of nodes).

A test case based on matrix multiplication is analyzed and the results obtained with the model are compared with the ones collected experimentally.

2 Regular SPMD applications

Many applications are suitable for the parallelization on regular topologies (e.g ring or 2-D mesh) with a even distribution of data among processors.

The code in each node consists normally of an initialization phase, a loop and a termination phase (Fig. 1). In each loop iteration there are a computation phase and a communication phase with neighbouring nodes, i.e. nodes connected by direct links on the considered topology.

For the generic l-th loop iteration ($l = 1, \ldots, L$), the elapsed time T_i on the i-th node can be expressed as

$$T_i = T_i^{comp} + T_i^{comm} + T_i^{idle} \tag{1}$$

```
P::

        initialization phase
        loop
           compute
           send data to neighbouring nodes
           receive data from neighbouring nodes
        end loop
        termination phase
```

Fig. 1. Process structure on each node

Usually, the send is asynchronous and the receive is blocking, resulting in a loosely synchronization among processes. Since we have a regular partitioning on a homogeneous parallel system, the application is self-balancing ($T_i^{idle} \simeq 0$).

Sometimes, depending on the particular application, we can achieve a more efficient implementation slightly modifying the loop structure, for example moving the data sending before the computation.

Communications can be carried out using proprietary primitives, optimized for the different architectures, but more often standard libraries such as PVM or MPI are used, ensuring code portability among different platforms.

This computational scheme occurs in various applications [6]. Among the others we mention matrix multiplication, long-range interactions [5], finite difference methods for the solution of Laplace equations. Other types of applications, such as finite element methods, particle dynamics and some kind of image processing [9] have a similar scheme but may require in addition the use of global communications and/or collective operations.

3   The heterogeneous computing environment

Let us consider an heterogeneous network of workstations or personal computers (generically denoted by NOW) consisting of p machines, in general with different features, connected by a switched communication network (e.g. Ethernet, Fast-Ethernet or ATM), with all links providing the same communication speed.

For a given application A, let us assume that the heterogeneity of nodes can be expresses by a single parameter, namely the relative speed s_i of node i with respect to a fixed reference machine, not necessarily belonging to the network [3]. s_i depends mainly on the clock speed ratio of nodes but also on the kind of application, cache and memory size and organization.

It is beyond the scope of this paper to provide a precise characterization of the node speed [14]. We suppose that speeds can be measured executing the application under investigation on the various nodes, or benchmarks belonging to the same class. We assume that the speed of each node does not vary, at least

as a first approximation, if we measure it using the whole application or any portion of it [8].

The total relative speed of the p node NOW is

$$S = \sum_i s_i \qquad (2)$$

and the average relative speed is

$$\bar{s} = \frac{S}{p} \qquad (3)$$

Since in the present work we are mainly interested in discussing the impact of heterogeneity, we suppose that the NOW is dedicated. Otherwise we can use an equivalent relative speed given by

$$s_i(1 - \sigma_i) \qquad (4)$$

σ_i being the load factor of node i.

Let us assume that transmission time along the network can be expressed as

$$t_{trans} = \alpha + \beta \cdot M \qquad (5)$$

where α is the latency (average value over the NOW), β is the communication time per byte and M is the message length in bytes.

We refer to message passing library such as PVM or MPI [12]. In this case the communication time for a message is the sum of three contributions [10]

$$t_{comm} = t_{pk} + t_{trans} + t_{upk} \qquad (6)$$

where t_{pk} is the time to prepare the message on the sending node, and t_{upk} is the time to unpack the message on the receiving node.

The time for packing/unpacking is greater if the encoding of data in a machine independent format is required; if all machines involved in communication support the same data format no encoding is needed, and t_{pk} and t_{upk} are greatly reduced.

To be exact, t_{pk} and t_{upk} depend on the speed of the node. However, since these terms are normally much smaller than t_{trans} and t_{comp}, we can. at least as a first approximation, neglect them or otherwise consider their average value over the nodes and add up it to t_{trans}. In both cases we assume communication speed constant for all nodes.

4 Performance analysis of SPMD applications on NOWs

Let W be the total computational work involved with the application A under consideration, and let τ be an atomic computing time (e.g. the time per element or per operation) on the reference node.

The application A is decomposed by data parallelization into p processes, each requiring a computational work W_i, and the process i-th is executed on node i-th.

The computation time for one out of L loop iterations on a generic node i-th is therefore

$$T_i^{comp} = \frac{W_i}{L}\frac{\tau}{s_i}, \quad i = 1, \dots, p \tag{7}$$

and $T_i^{comm} = T^{comm}$ is the corresponding communication time, which under the assumptions of the previous section is the same for all nodes.

Dealing with heterogeneous computing systems, we are mainly interested in evaluating the idle time on each node, since this is the main factor that can lower the overall performance.

Let us consider an application with processes connected on a logical ring. Let us define

$$\Delta T_i^{comp} = T_j^{comp} - T_i^{comp} = (\frac{W_j}{s_j} - \frac{W_i}{s_i})\frac{\tau}{L}, \quad i = 1, \dots, p \tag{8}$$

where j denotes the node with the highest computation time.

From our analysis it turns out that we can have two different behaviours, depending on computation and communication times and on the degree of heterogeneity of the network. More precisely, it exists a threshold value for T^{comm}

$$T_{th}^{comm} = \frac{\sum_i \Delta T_i^{comp}}{p} \tag{9}$$

which allows to distinguish the two following situations.

a) If $T^{comm} \leq T_{th}^{comm}$, after a transient phase of p iterations, a steady state is reached, characterized by the fact that the idle time of each node does not vary from an iteration to another, and it is given by

$$T_i^{idle} = \Delta T_i^{comp}, \quad i = 1, \dots, p \tag{10}$$

The duration of the transient phase does not depend on the mapping of processes to nodes.

b) If $T^{comm} > T_{th}^{comm}$, the situation is slightly more involved since after the transient phase we get a periodic behaviour (with period p) where the average idle time over the set of nodes for each loop iteration is equal to T^{comm}.

Similar considerations apply for mesh based applications, but the duration of the transient phase can depend on the mapping.

In the following we will deal with case a), since in practice performance is limited by unbalancing. Of course, improving load balancing we move from case a) to case b); however, in the b) situation, performance cannot further be improved, unless we modify the algorithm.

To obtain a first approximation of the impact of heterogeneity on efficiency we can neglect the communication overhead ($T_i^{comm} = 0$). Following [3], and using eqs. (10,8) the node-level efficiency is

$$\eta_i = \frac{T_i^{comp}}{T_i^{comp} + T_i^{idle}} = \frac{T_i^{comp}}{T_j^{comp}}, \quad i = 1, \dots, p \tag{11}$$

and the global efficiency is

$$\eta = \frac{\sum_i s_i \eta_i}{\sum_i s_i} \tag{12}$$

Of course, the efficiency above is an upper bound of the actual efficiency, since we have neglected the communication overhead. η_i and η are at the step level, but they coincide with the efficiencies of the whole computation, since all the loop iterations are equal.

4.1 Evaluating unbalancing for naive porting

In the case of a straightforward porting by homogeneous data partitioning of a regular application on a heterogeneous NOW we have

$$W_i = \frac{W}{p}, \quad i = 1, \dots, p \tag{13}$$

Therefore eq. (8) becomes

$$\Delta T_i^{comp} = \left(\frac{1}{s_j} - \frac{1}{s_i} \right) \frac{W}{p} \frac{\tau}{L}, \quad i = 1, \dots, p \tag{14}$$

and eq. (9) particularizes to

$$T_{th}^{comm} = \left(\frac{1}{s_j} - \bar{s}_H \right) \frac{W}{p} \frac{\tau}{L} \tag{15}$$

where $\bar{s}_H$ denotes the harmonic mean over s_i.

As expected, there is no idle time on the slowest node (in this case the node with the highest computation time is the slowest one), and the idle time increases with the node speed. Since the fastest nodes are poorly exploited, the global efficiency is low.

Using eqs. (7,13) we obtain

$$\eta_i = \frac{s_j}{s_i}, \quad i = 1, \dots, p \tag{16}$$

and

$$\eta = \frac{s_j}{\bar{s}} \tag{17}$$

where $\bar{s}$ is the average relative speed over the nodes composing the NOW.

4.2   Strategies to improve efficiency

Eqs. (10.8) show that the computational work carried out by each node should be as proportional as possible to its relative speed in order to keep unbalancing low, i.e.

$$W_i \simeq \frac{s_i}{S} W, \quad i = 1, \ldots, p \tag{18}$$

This can be achieved in two ways.

1) By using an heterogeneous partitioning of data among processors. With this approach some changes in the code are required, thus making the porting more costly. It may be useful to employ semi-automatic tools such as that proposed in [2].

2) By splitting homogeneously the application in a number of processes q greater than the number of nodes p, and assigning to each node a number of processes q_i as proportional as possible to its relative speed, i.e.

$$q_i \simeq \frac{s_i}{S} q, \quad i = 1, \ldots, p \tag{19}$$

Of course, to maximize performance it is convenient to put logically neighbouring processes on the same physical node [4].

In the remaining part of this section we address in more detail the second approach. In this case, the computational work for each node is

$$W_i = \frac{q_i}{q} W, \quad i = 1, \ldots, p \tag{20}$$

The node-level and global efficiencies become

$$\eta_i = \frac{q_i}{q_j} \frac{s_j}{s_i}, \quad i = 1, \ldots, p \tag{21}$$

and

$$\eta = \frac{q}{q_j} \frac{s_j}{S} \tag{22}$$

We see that local efficiencies increase if the ratios q_i/q_j are close to the corresponding ratios s_i/s_j. Moreover, for a fixed q, η is maximum if $q_j/s_j = \max_i(q_i/s_i)$ is minimum.

The number of processes required to achieve a good balancing increases with the degree of heterogeneity of the network, and correspondingly the process granularity must decrease.

To show an example of application, let us consider a NOW of six nodes, with constant total power $S = 6$, and therefore $\bar{s} = 1$. We choose the four different configurations with increasing heterogeneity (expressed by $h = 1 - s_{\min}/\bar{s}$, where $s_{\min}$ denotes the lowest relative speed in the NOW) reported in Table 1, and we vary the number of total processes from 6 up to 75. For each configuration and each value of q we find the optimal q_i using the criterion of minimizing q_j/s_j.

Table 1. Configurations with increasing heterogeneity used in Fig. 2; $p = 6$; $S = 6$

s_i						h
0.6	0.8	1.0	1.0	1.2	1.4	0.40
0.4	0.6	0.8	1.2	1.4	1.6	0.60
0.2	0.2	0.6	0.6	2.2	2.2	0.80
0.1	0.4	0.7	1.3	1.6	1.9	0.90

Fig. 2 shows the global efficiency, computed from eq. (22), versus the degree of heterogeneity of the system. We see that, as expected, for a fixed q efficiency decreases as h increases, and the effect is stronger when q is smaller. However, a reasonable number of processes (e.g. $q \simeq 50$) is sufficient to achieve an efficiency of about 0.8, also with a highly heterogeneous network.
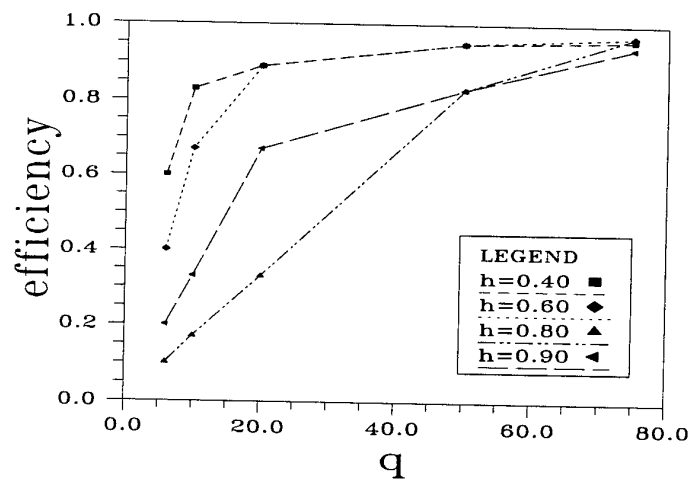
Fig. 2. Efficiency vs. the total number of processes q, for various values of h; $p = 6$; $S = 6$.

Of course, a trade-off exists between balancing and the need of keeping low other overheads, in particular the time lost due to context switching, which is not considered in the present analysis.

The approach based on the use of multiple processes per node permits a complete reuse of code developed for homogeneous platforms.

5   Simulation and experimental results

We set up a simple model able to simulate the execution of regular applications on NOWs, with the three different approaches outlined in the previous section. The model uses some parameters at the hardware level (i.e. the number of processors p and the network speed, expressed by α and β), and some parameters which also depend on the selected application (i.e. the atomic time τ on the reference node and the relative node speeds s_i). The third approach also requires the total number of processes q. From such low level parameters, the model computes for the given application the computation, communication and idle times at the loop iteration level for each processor. In this way the model yields the figures of speed-up and efficiency of the whole application.

The model is tested using the matrix multiplication algorithm that computes $C = A \times B$, with A, B and C $n \times n$ matrices, on a logical ring of processes, as described in [13].

In the original SPMD implementation with homogeneous data partitioning each processor i stores a slice of matrix A and a slice of matrix B, each comprising rows from $(i-1)n/p$ to $i \cdot n/p$. Slices of A remain local to the various processors, whereas slices of B circulate along the ring. The whole computation requires p loop iterations and at the end processor i has computed n/p rows of C, from row $(i-1)n/p$ to row $i \cdot n/p$.

So, the computation time of node i during the l-th iteration is

$$T_i^{comp} = 2\frac{n^3}{p^2}\frac{\tau}{s_i}, \quad i = 1, \ldots, p \tag{23}$$

and in each iteration n^2/p elements of B are moved between neighbouring nodes.

Using the heterogeneous data partitioning approach means in this case to assign slices of matrix A to each node with a number of rows proportional to its relative speed, whereas matrix B is still evenly partitioned among nodes.

The third approach is exactly the same as the first, with the exception that q processes (with $q > p$) are generated and the optimal q_i are given by eq. (19).

The various versions of this test program are implemented using C language and PVM v. 3.4 and executed on a variable number of nodes belonging to a NOW of six workstations connected by a switched Ethernet. Table 2 shows the characteristics of the various nodes and the total power of the different configurations.

The trials are executed on dedicated nodes and with a low traffic on the network. The measured value of the time per element on the reference node is $\tau = 0.56\ \mu sec$. We measure on the network the values $\alpha = 1\ msec$ and $\beta \simeq 1\ \mu sec$.

Experimental data has been collected using 1000×1000 floating point matrices. Table 3 reports the measured and simulated speed-up for the three different approaches. As expected, the speed-up of the straightforward homogeneous partitioning is well below the ideal one, while the two proposed strategies to reduce unbalancing yield considerably better speed-up figures.

Table 2. The first column identifies the configuration, that includes nodes up to the current row; for each configuration the type and the relative speed of the nodes, the total computing power and the degree of heterogeneity h are reported

Config. Id.	Workstation	Relative speeds	Available computing power	h
-	Sparc-20	1.00	1.00	-
C1	SGI-O2	1.87	2.87	0.31
C2	SGI-O2	1.90	4.77	0.37
C3	Sparc-Ultra 5	1.87	6.64	0.40
C4	Sparc-Ultra 5	1.85	8.49	0.41
C5	Indigo 2	5.87	14.36	0.58

Table 3. The first column gives the configuration identifier; the SPMD columns provide speed-up for homogeneous SPMD application measured (M) and simulated(S); HD columns summarize speed-up for heterogeneous data partitioning; the VP colums provide speed-up for homogeneous data partitioning but with a number of processes on each node proportional to its relative speed (the total number of processes q is reported in the last column)

Config. Id.	SPMD-M	SPMD-S	HD-M	HD-S	VP-M	VP-S	q
C1	2.06	1.99	2.89	2.86	2.71	2.80	3
C2	3.09	3.00	4.69	4.76	4.68	4.66	5
C3	4.62	4.00	6.62	6.62	6.14	6.51	7
C4	5.80	5.00	8.36	8.48	8.18	8.35	9
C5	6.54	5.95	13.23	14.24	12.67	13.6	15

Measured and experimental data are in most cases in good agreement, thus confirming that the proposed model is quite reliable. The maximum errors occur in the case of SPMD homogeneous implementation, and it is due to an under-estimation of the relative speed of the slowest nodes. In fact we assume that the relative speed of each node does not vary with the data size handled by the node. Indeed, we can sometimes observe a gain in processor speed when the amount of local data decreases, for example due to better use of the hierarchy of memories. This is more relevant in the homogeneous data partitioning case where the relative weight of the slowest nodes is greater.

6   Conclusions

We analized the problem of porting data-parallel applications originally developed for homogeneous parallel systems with regular topologies (e.g. ring or mesh) to network of workstations and/or personal computers.

For this kind of computing resources, maintaining the even partitioning of data among processors yields poor performance, since efficiency is limited by unbalancing, that increases with the degree of heterogeneity of the network.

Two strategies are considered to overcome this problem: heterogeneous data partitioning or allocation to each node of a number of processes proportionally to its relative power.

A simple model is proposed to analyze and predict performance of the considered class of applications using the various approaches.

The model is tested using a matrix multiplication algorithm with processes arranged in a ring topology. A good agreement is obtained between simulated and experimental figures of performance both for the naive unbalanced implementation and for the two improved implementations.

References

1. Beaumont, O., Boudet, V., Rastello, F., Robert, Y.: Data Allocation Strategies for Dense Linear Algebra Kernels on Heterogeneous Two-dimensional Grids. Int. Parallel and Distributed Processing Symposium (IPDPS'2000). Cancun (Mexico), 1-5 May 2000
2. Cermele, M., Colajanni, M., Necci, G.: Dynamic Load Balancing of Distributed SPMD Computations with Explicit Message Passing Systems. Proc. 6-th Heterogeneous Computing Workshop. IEEE CS Press (1997) 2-16
3. Clematis, A., Corana, A.: Modeling Performance of Heterogeneous Parallel Computers. Parallel Computing **25** (1999) 1131-1145
4. Clematis, A., Dodero, G., Gianuzzi, V.: A Resource Management Tool for Heterogeneous Networks. Proc. 7-th Euromicro Workshop on Parallel and Distributed Processing. IEEE CS Press (1999) 367-373
5. Corana, A.: Parallel Computation of the Correlation Dimension from a Time Series. Parallel Computing **25** (1999) 639-666
6. Fox, G.C., Johnson, M.A., Lyzenga, G.A., Otto, S.W., Salmon, J.K., Walker, D.W.: Solving Problems on Concurrent Processors. Vol. 1. Prentice-Hall, Englewood Cliffs, NJ (1988)
7. Khokhar, A.A., Prasanna, V.K., Shaaban, M.E., Wang, C.: Heterogeneous Computing: Challenges and Opportunities. Computer **26** (1993) 18-27
8. Mazzeo, A., Mazzocca, N., Villano, U.: Efficiency Measurements in Heterogeneous Distributed Computing Systems: from Theory to Practice. Concurrency: Practice and Experience **10** (1998) 285-313
9. Ranka, S., Sahni, S.: Hypercube Algorithms with Applications to Image Processing and Pattern Recognition. Springer-Verlag, Berlin Heidelberg New York (1990)
10. Schmidt, B.K., Sunderam, V.S.: Empirical Analysis of Overheads in Cluster Environments. Concurrency, Practice and Experience **6** (1994) 1-32

11. Sunderam, V.S.: Methodologies and Systems for Heterogeneous Concurrent Computing. In: Joubert, G.R., Trystram, D., Peters, F.J., Evans, D.J. (eds.): Parallel Computing: Trends and Applications. Elsevier, Amsterdam (1994) 29-45
12. Sunderam, V.S., Geist, G.A., Dongarra, J., Manchek, R.: The PVM Concurrent Computing System: Evolution, Experiences, and Trends. Parallel Computing **20** (1994) 531-545
13. Yan, Y., Zhang, X., Song, Y.: An Effective and Practical Performance Prediction Model for Parallel Computing on Nondedicated Heterogeneous NOW. J. Parallel and Distributed Computing **38** (1996) 63-80
14. Zaki, M.J., Li, W., Cierniak, M.: Performance Impact of Processor and Memory Heterogeneity in a Network of Machines. Proc. Fourth Heterogeneous Computing Workshop. Santa Barbara, California (1995)

Parallelization of a Recursive Decoupling Method for Solving Tridiagonal Linear System on Distributed Memory Computer

M. Amor[1], F. Argüello[2], J. López[3], and E. L. Zapata[3]

[1] Department of Electronics and Systems, University of A Coruña,
E–15071 La Coruña, Spain
`margaaml@udc.es`
[2] Department of Electronics and Computation, University of Santiago de Compostela,
E–15706 Santiago de Compostela, Spain
`arguello@dec.usc.es`
[3] Department of Computer Architecture, University of Málaga
E–29071 Málaga, Spain
`{juan,ezapata}@ac.uma.es`

Abstract. This work presents a parallelization of a recursive decoupling method for solving tridiagonal linear system on distributed memory computer. We study the fill-in in the algorithm to optimize the execution of the scalar algorithm and to perform the communications. Finally, we evaluate the algorithm through specific test on the Fujitsu AP3000.

1 Introduction

In recent years considerable effort has been devoted to solve tridiagonal systems (TS), a very important class of linear systems which appear when the finite differential method is used to solve differential equations in partial derivates such as simple harmonic motion, Helmhotz, Poisson, Laplace and diffusion equations. The finite differential method involves the discretization of the differential equation and subsequently the solution of the tridiagonal systems thus generated.

There are many algorithms for solving TS, such as Gaussian elimination or LU elimination, that have proved to be the most effective sequential algorithms on serial computers. However, these algorithms cannot be directly adopted to parallel computers. Much research has been undertaken on parallel algorithms for solving TS. Hockney proposed the *cyclic (odd-even) reduction* (CR) algorithm in 1965. Although originally proposed as sequential, this algorithm can be adapted to run on a wide range of parallel architectures [8, 5]. In addition, new methods for increasing the parallelism of CR algorithm, such as PARACR [9] or radix-p CR algorithm [8], have been proposed. On the other hand, other well known strategies have been adapted to get new TS parallel algorithms, such as the proposed by Eğecioğlu et al. [6] (recursive doubling strategy), Lin and Cheng [12](prefix), and Wang and Mou [17] and, Spaletta and Evans [16], which exploit

the parallelism of the divide-and-conquer strategy. Finally, a group of hybrid algorithms have been proposed that are based on partitioning the system into blocks of equations, using a local algorithm to reduce the subsystem in each block and a global algorithm to solve the reduced system. In this group we include the algorithms by Krechel, Plum and Stüben [10], Cox and Knisley [4], Müller and Scheerer [15], Matton, Williams and Hewett [14] and Amodio and Brugnano [2]. In [1] we have classified the above TS algorithms in terms of their data flows and presented a unified parallelization on computers with mesh topology and distributed memory.

In this paper, we consider the parallelization of the recursive decoupling algorithm by Spaletta and Evans [16] on a distributed memory multiprocessor. This algorithm has a very good behavior in terms of accuracy as the problem size increases and the partitioning process leads to independent systems. As stablished in previous works, the memory allocation requirement is demanding [16] and the execution times are not competitive with other partitioning methods [1]. In this paper we propose a technique to reduce the execution time of the scalar algorithm, minimize the memory requirements and to optimize the communications in the parallel implementation. This technique is based in the sparsity of the matrix obtained in the recursive fill-in process of this algorithm.

The rest of the work is organized as follows: in Section 2 we present the recursive decoupling algorithm by Spaletta-Evans. The parallel algorithm is presented in Section 3. Experimental results on the Fujitsu AP3000 multiprocessor are shown in Section 4. Finally, in Section 5 we present the conclusions.

2   The Recursive Decoupling Algorithm

We consider a set of N linear equations with N unknowns

$$A\mathbf{u} = \mathbf{d}, \tag{1}$$

where A is a tridiagonal matrix $N \times N$ of the form

$$A = \begin{pmatrix} b_0 & c_0 & & & & \\ a_1 & b_1 & c_1 & & & \\ & a_2 & b_2 & c_2 & & \\ & & \cdot & \cdot & \cdot & \\ & & & \cdot & \cdot & \cdot \\ & & & a_{N-2} & b_{N-2} & c_{N-2} \\ & & & & a_{N-1} & b_{N-1} \end{pmatrix}, \text{with } |b_i| \geq |a_i| + |c_i|, \ \forall i = 0, 1, ..., N-1. \tag{2}$$

With no loss of generality we will assume that the number of equations is a power of two. We will denote $m = N/2 = 2^{n-1}$.

The recursive decoupling algorithm is based in the recursive calculation of the inverse of matrix A by means of the Sherman–Morrison formula [7]. To this

goal, we decompose the matrix A (2) as follows:

$$\begin{pmatrix} e_0 & c_0 & & & & \\ a_1 & e_1 & & & \mathbf{0} & \\ & & e_2 & c_2 & & \\ & & a_3 & e_3 & & \\ & & & & \ddots & \\ \mathbf{0} & & & & e_{N-2} & c_{N-2} \\ & & & & a_{N-1} & e_{N-1} \end{pmatrix} + \sum_{j=1}^{m-1} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{N-2} \\ x_{N-1} \end{pmatrix}^{(j)} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{N-2} \\ y_{N-1} \end{pmatrix}^{(j)T} \tag{3}$$

where

$$\begin{aligned} e_0 &= b_0 \\ e_{N-1} &= b_{N-1} \end{aligned} \tag{4}$$

and

$$\left. \begin{aligned} e_{2j-1} &= b_{2j-1} - a_{2j} \\ e_{2j} &= b_{2j} - c_{2j-1} \end{aligned} \right\} \quad \text{when } j = 1, \dots, m-1. \tag{5}$$

In expression (3), all the elements in the vector columns $\mathbf{x}^{(j)}$ and $\mathbf{y}^{(j)}$ have only two non–zero elements at the positions $2j - 1$ and $2j$, that is

$$\mathbf{x}^{(j)} = (0, \cdots, 0, 1, 1, 0, \cdots, 0)^T \tag{6}$$
$$\mathbf{y}^{(j)} = (0, \cdots, 0, a_{2j}, c_{2j-1}, 0, \cdots, 0)^T$$

In matrix notation, the partitioning of A given in equation (3) is denoted as

$$A = J + \sum_{j=1}^{m-1} \mathbf{x}^{(j)} \mathbf{y}^{(j)T}, \tag{7}$$

where J is the 2×2 block diagonal matrix on the left in equation (3).

The basic idea, underlying the choice of this particular partitioning, is given by the Sherman–Morrison method. Sherman–Morrison proved that, given two $N \times N$ matrices A and J such that $A = J + x \cdot y^T$, the inverse of matrix A can be obtained by the formula

$$A^{-1} = J^{-1} - \alpha(J^{-1}\mathbf{x})(\mathbf{y}^T J^{-1}), \ \alpha = \frac{1}{1+\mathbf{y}^T J^{-1}\mathbf{x}}. \tag{8}$$

To directly compute the inverse of matrix A would cost $O(N^3)$ arithmetic operations, while the use of formula (8) only implies $O(N^2)$ operations. When applied to solve a linear system of equations $A\mathbf{u} = \mathbf{d}$, the solution will be

$$\mathbf{u} = A^{-1}\mathbf{d} = (I - \alpha J^{-1}\mathbf{x}\mathbf{y}^T)J^{-1}\mathbf{d}. \tag{9}$$

This process avoids the explicit computation of the inverse matrix.

The Recursive Decoupling method, described in [16], derives the solution of system (1) by considering that $A = J + \sum_{j=1}^{m-2} \mathbf{x}^{(j)} \mathbf{y}^{(j)T} + \mathbf{x}^{(m-1)} \mathbf{y}^{(m-1)T}$, then

applying the Sherman–Morrison formula (7) to matrices A and $J + \sum_{j=1}^{m-2} \mathbf{x}^{(j)} \mathbf{y}^{(j)T}$. The recursive procedure is as follows

$$
\begin{aligned}
\mathbf{M_h} &= \left(J + \sum_{j=1}^{h} \mathbf{x}^{(j)} \mathbf{y}^{(j)T} \right)^{-1} = \left(I - \alpha_{h-1} \mathbf{M}_{h-1} \mathbf{x}^{(h-1)} \mathbf{y}^{(h-1)T} \right) \mathbf{M}_{h-2} \\
\alpha_{h-1} &= 1 / \left(1 + \mathbf{y}^{(h-1)T} \mathbf{M}_{h-1} \mathbf{x}^{(h-1)} \right).
\end{aligned}
\tag{10}
$$

Index h goes from 1 to $m - 1$, M_0 being the matrix J^{-1} and the last matrix M_{m-1} will be A^{-1}. Let us denote as $g^{(h-1)} = M_{h-1} x_h$. Observe that these vectors are needed to obtain the recursive formula (10) and can be computed using a similar recursive method

$$
\begin{aligned}
\mathbf{g^{(h)}} &= \left(I - \alpha_{h-1} \mathbf{g}^{(h-1)} \mathbf{y}^{(h-1)T} \right) \mathbf{M}_{h-2} \mathbf{x}_h \\
&= \left[\prod_{j=1}^{h-1} (I - \alpha_j \mathbf{g}^{(j)} \mathbf{y}^{(j)T}) \right] \mathbf{J}^{-1} \mathbf{x}_h.
\end{aligned}
\tag{11}
$$

In order to obtain the final solution $\mathbf{u} = A^{-1} \mathbf{d}$, from (10) follows a recursive formula similar to (11)

$$
\mathbf{u} = \mathbf{A}^{-1} \mathbf{d} = \left[\prod_{j=1}^{m-1} (I - \alpha_j \mathbf{g}^{(j)} \mathbf{y}^{(j)T}) \right] \mathbf{J}^{-1} \mathbf{d}.
\tag{12}
$$

Then we need to carry out the following steps,

step 1 In this step the matrix J^{-1} is calculated, as well as the product $\mathbf{J}^{-1} \mathbf{d}$, the initial value of $\mathbf{u}$. Given the shape of matrix J, its inverse may be obtained by calculating the inverse of each 2×2 block J_j,

$$
J_j^{-1} = \frac{1}{\Delta_j} \begin{pmatrix} e_{2j-1} & -c_{2(j-1)} \\ -a_{2j-1} & e_{2(j-1)} \end{pmatrix}, \quad \Delta_j = e_{2(j-1)} e_{2j-1} - a_{2j-1} c_{2(j-1)}.
\tag{13}
$$

so, the value of $\mathbf{J}^{-1} \mathbf{d}$ becomes

$$
\mathbf{J}^{-1} \mathbf{d} = \begin{pmatrix}
(e_1 d_0 - c_0 d_1)/\Delta_1 \\
(-a_1 d_0 + e_0 d_1)/\Delta_1 \\
\vdots \\
(e_{2j-1} d_{2(j-1)} - c_{2(j-1)} d_{2j-1})/\Delta_j \\
(-a_{2j-1} d_{2(j-1)} + e_{2(j-1)} d_{2j-1})/\Delta_j \\
\vdots \\
(e_{2m-1} d_{2m-2} - c_{2m-2} d_{2m-1})/\Delta_m \\
(-a_{2m-1} d_{2m-2} + e_{2m-2} d_{2m-1})/\Delta_m
\end{pmatrix}.
\tag{14}
$$

Step 2 Compute the initial vectors $\mathbf{g}^{(j)} = \mathbf{J}^{-1}\mathbf{x}^{(j)}$ for indices $j = 1, \cdots, m - 1$. Because the pattern of the vector $\mathbf{x}^{(j)}$, the vector $\mathbf{g}^{(j)}$ has only non–zero elements from $2j - 1$ to $2j + 2$ positions,

$$\mathbf{g}^{(j)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ -c_{2(j-1)}/\Delta_{2j-1} \\ e_{2(j-1)}/\Delta_{2j-1} \\ e_{2j-1}/\Delta_{2j} \\ -a_{2j-1}/\Delta_{2j} \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \tag{15}$$

Step 3 In this last stage, vectors $\mathbf{u}$ y $\mathbf{g}^{(j)}$ are updated by using the equations (11) and (12). This rank-one updating procedure, which also make use of the particular shape of vectors $\mathbf{x}^{(j)}$ and $\mathbf{y}^{(j)}$, can be described as follows:

$$
\begin{aligned}
&\text{for } k = 1, 2, \cdots, n - 1 \\
&\quad \text{for } j = 2^{k-1}, \, 2^{n-1} - 2^{k-1}, \, 2^k \\
&\qquad \alpha_j = 1/(1 + \mathbf{y}^{(j)T}\mathbf{g}^{(j)}) \\
&\qquad \mathbf{u} = \left(I - \alpha_j \mathbf{g}^{(j)}\mathbf{y}^{(j)T}\right)\mathbf{u} \\
&\qquad \text{for } i = 2^k, \, 2^{n-1} - 2^k, \, 2^k \\
&\qquad\quad \mathbf{g}^{(i)} = \left(I - \alpha_j \mathbf{g}^{(j)}\mathbf{y}^{(j)T}\right)\mathbf{g}^{(i)} \\
&\qquad \text{end} \\
&\quad \text{end} \\
&\text{end}
\end{aligned}
$$

3   The Parallel recursive decoupling algorithm

In this section we propose some modifications to the above sequential algorithm in order to reduce storage and execution time. Then, we propose a parallelization of the algorithm.

Note that in step 2, when we calculate $\mathbf{g}^{(j)}$, $(0 \leq j \leq m-1)$, the initial vectors $\mathbf{x}^{(j)}$ only contain 2 non–zero elements. Therefore, at the 1st iteration the vectors $\mathbf{g}^{(j)}$ are composed of 4 non–zero elements and, in general, at iteration k, $\mathbf{g}^{(j)}$ is a vector with 2^{k+1} non–zero elements, namely components from $2j + 1 - 2^k$ to $2j + 2^k$.

Observe at the example in Fig. 1 that to compute vector $\mathbf{g}^{(i)}$ we do not need all the $\mathbf{g}^{(j)}$ vectors in each iteration k. In fact, only are needed those vectors $\mathbf{g}^{(j)}$ which have elements different from 0 just at row i, where column i of matrix $\left(I - \alpha_j \mathbf{g}^{(j)} 2\mathbf{y}^{(j)T}\right)$ has also elements different from 0. It can be easily proved that this happens if $2^{k+1}\lfloor \frac{j}{2^{k+1}} \rfloor \leq i \leq j + 2^{k-1}$. Then, the internal loop i in the step 3 of the recursive procedure can be simplify as follows

$\text{begin} = 2^{k+1} \lfloor \frac{j}{2^{k+1}} \rfloor$

if(begin == 0) then begin $= 2^k$

for $i = begin; j + 2^{k-1}, 2^k$

$$\mathbf{g}^{(i)} = \left(I - \alpha_j \mathbf{g}^{(j)} \mathbf{y}^{(j)T}\right) \mathbf{g}^{(i)}$$

end

On the other hand, the fill-in process only occurs at several points of the algorithm, where the values associated to specifics $\mathbf{g}^{(i)}$ are computed. These values are calculated using the recursive tree procedure described in step 3. For example, in Fig. 2, at the iteration 1, j has the values $\{1, 3, 5, 7\}$, at the iteration 2 has the values $\{2, 6\}$ and at the iteration 3 has the value $\{4\}$. In addition, the vectors $\mathbf{g}^{(1)}$, $\mathbf{g}^{(3)}$, $\mathbf{g}^{(5)}$ and $\mathbf{g}^{(7)}$ are used only at the 1st iteration and during the execution of the algorithm keep at most 4 non–zero elements. Similarly, vectors $\mathbf{g}^{(2)}$ and $\mathbf{g}^{(6)}$ are used until the 2nd iteration and the number of non–zero elements is less than 8, and so on. As a consequence, not all the vectors $\mathbf{g}^{(i)}$ perform the fill-in procedure in the same way. We take advantage of this fact to gather the non–zero elements, then saving memory. Instead of arrays $g[\ ][\ ]$ of size $N/2 \times N/2$, we have arrays of size $(n-1) \times N/2$. At the stage 2 in Fig. 1 we can see how the vectors $\mathbf{g}^{(j)}$ are stored for the case $N = 16$. Memory savings is $(2^{n-1} - n + 1)2^{n-1}$.

Concerning the parallelization of the algorithm, Fig. 1 summarizes their stages by means of an example ($N = 16$ equations on 4 PEs). In this algorithm, the responsibility to perform the computation of the initial steps is divide among all the processors. Therefore, the process of partitioning matrix A, given in (7), as well as the distribution of vectors $\mathbf{u}$ and $\mathbf{d}$ is refered as *preliminary stage*. At this stage, communications of the $c_{(N \cdot i)/P - 1}$ occur from processor i to processor $i + 1$ and, for the $a_{(N \cdot i)/P}$, from processor $i + 1$ to processor i, where $i = 1, \cdots, P - 1$, P being the number of processors (see Fig. 1).

After *preliminary stage*, the steps 1 to 3 are computed. Having in mind the block diagonal structure of matrix J, step 1 may be computed concurrently in all the processors without any communication, since the m subsystems in (13) can be solved in parallel. The same happens at stage 2, but in this case the $m - 1$ subsystems in (15) are to be solved. Some vectors $\mathbf{g}^{(j)}$ are distributed among two processors. But this does not imply any comunication since each processor calculates the components of the vector $\mathbf{g}^{(j)}$ using local data. As an example, in Fig.1, the components $\{2, 3\}$ of vector $\mathbf{g}^{(2)}$ are in processor 0 and the elements $\{4, 5\}$ in processor 1. This distribution of vector $\mathbf{g}^{(j)}$ provides a better load balance.

At the stage 3 no communications are required during the first $n - p - 1$ iterations. However, the last p iterations require communications since the $i - th$ element of vector $\mathbf{u}$ must be transfered to all the processors containing elements of the $i - th$ column of the matrix $\left(I - \alpha_j \mathbf{g}^{(j)} \mathbf{y}^{(j)T}\right)$ which are different from 0. In addition, the $k - th$ element of vector $g^{(j)}$ must be transfered to processors which contain elements of column k of $\left(I - \alpha_j \mathbf{g}^{(j)} \mathbf{y}^{(j)T}\right)$ differents from 0.
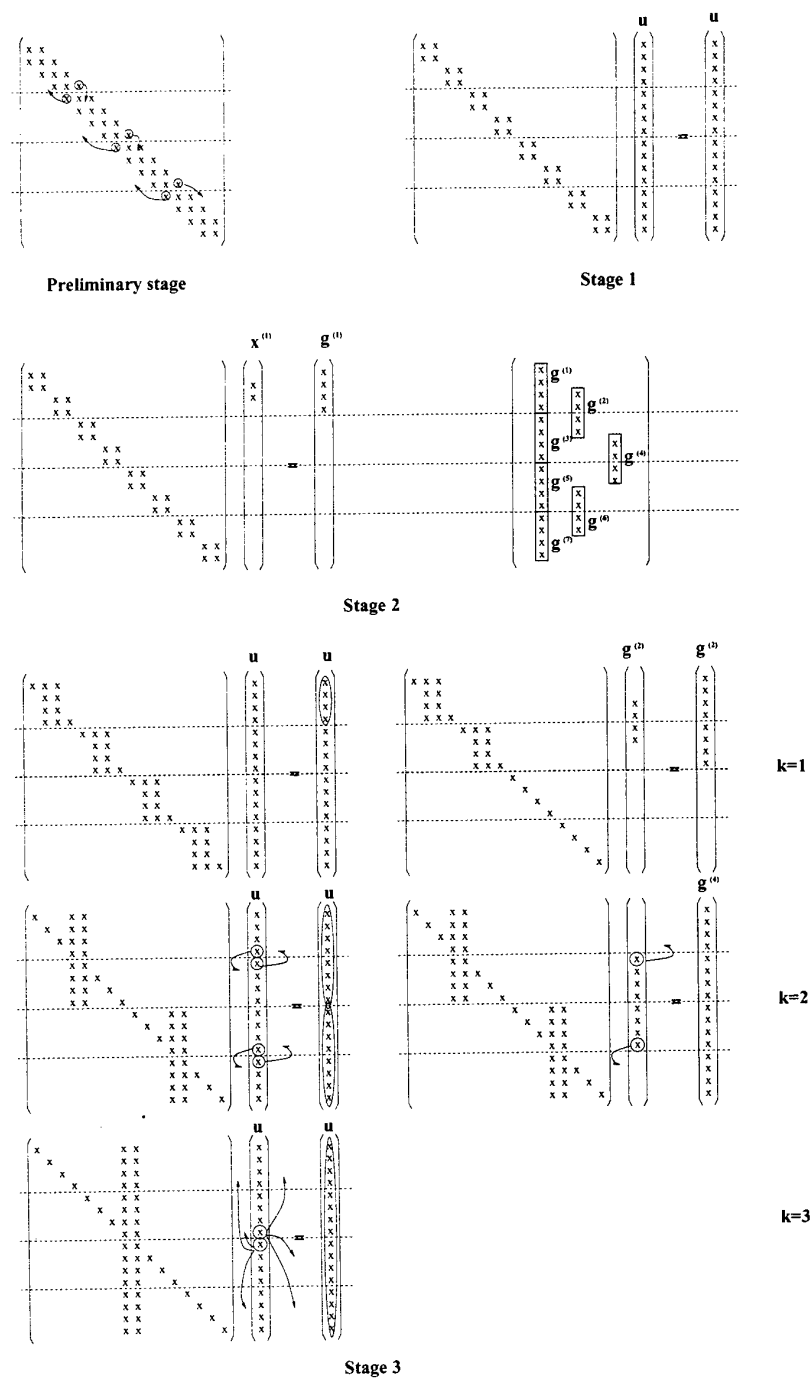
Fig. 1. Scheme of the parallel algorithm for $N = 16$ equations and 4 processors. We denote as $\times$ the elements differents from 0 either in vectors and matrices. Circles indicate data to be transfered and arrows point out destination processors. At stage 2, computation of $\mathbf{g}^{(1)}$ from $\mathbf{x}^{(1)}$ and g is summarized. At stage 3, the Figure shows how $\mathbf{g}^{(2)}$ for $k = 1$ and $\mathbf{g}^{(4)}$ $k = 2$ are calculated.
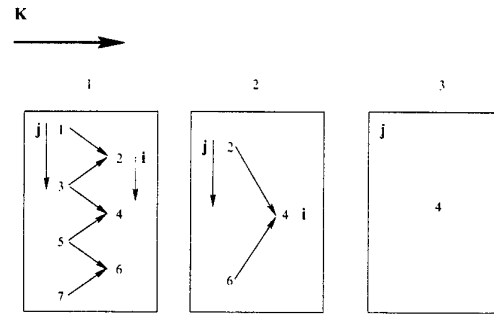
Fig. 2. the vectors $\mathbf{g}^{(j)}$ are calculated at each iteration of step 3 for $N = 16$ equations.

4 Evaluation

The recursive decoupling algorithm has been implemented on the Fujitsu AP3000 distributed memory computer [13] using the message passing programming model. We have used the MPI programming environment. To verify the performance of the parallel algorithm, we used a test diagonal system (with know solution), whose coefficients matrices satisfy the condition, $|b_i| \geq |a_i| + |c_i|$, $\forall i = 0, 1, ..., N - 1$. This test is described below,

$$\begin{pmatrix} 2 & -1 & & & & & \\ -1 & 2 & -1 & & & & \\ & -1 & 2 & -1 & & & \\ & & \cdot & \cdot & \cdot & & \\ & & & \cdot & \cdot & \cdot & \\ & & & & -1 & 2 & -1 \\ & & & & & -1 & 2 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix} \tag{16}$$

whose exact solution is an N-dimensional vector $\mathbf{u}$ with components:

$$u_i = \frac{N + 1 - i}{N + 1}, \ \forall i = 1, \cdots, N. \tag{17}$$

The experiments were performed on matrices of size ranging from 16384 (2^{14}) to 1048576 (2^{20}) for the test (16). As we can see in Table 1, the increasing number of processors produces a reduction in the execution time of the algorithm. We observe that this method presents a high efficiency for all the sizes of equations.

Fig. 3 depicts the experimental results. So, in Fig. 3.a we show the efficiency of the modified sequential algorithm we propose related to the initial algorithm efficiency. Thus, Observe than performance increases more than 91% for any value of N. On the other hand, in Fig. 3.b we show the efficiency for the parallel algorihtm for some values of parameter N. Efficiency was calculated using the execution time of the sequential code. The parallel algorithm exceeds the ideal speedup due to an efficient use of local memories and the communication

Table 1. Execution times in seconds measured on the AP3000 for differents number of processors. The size of matrices are from 16384 (2^{14}) to 1048576 (2^{20}).

P	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
1	0.4231	0.9098	1.9969	4.3576	9.2092	19.3657
2	0.1830	0.3561	0.7990	1.7162	3.9248	7.7731
4	0.0848	0.1866	0.3813	0.8379	1.9117	3.9418
8	0.0427	0.0897	0.1897	0.3988	0.9471	1.9001

optimization. Therefore, these results prove that the techniques employed to parallelize the algorithm permit to obtain a good performance on distributed memory computers. A last observation is that our parallel program is scalable. That is, in order to maintain a constant efficiency, N grow at the same rate as P, which we just observed in Fig. 3.b.

n   **Number of processors**
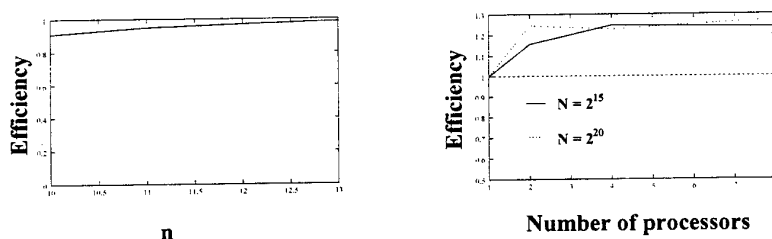
Fig. 3. (a) Efficiency of the modified sequential algorithm we propose related to the initial algorithm efficiency. (b) Efficiency of the parallel algorithm on the AP3000 for $N = 2^{15}$ and $N = 2^{20rad}$ data.

It is difficult to make a comparison with other implementations of the Recursive Decoupling Method for Solving Tridiagonal on other machines, but the speedup may be compared with the presented in [16, 3]. Their numerical results are obtained in the Balance 8000 multiprocessor system. The maximum speedup is 2.1075 with $N = 512$ and $P = 8$. Climent et al. [3] present theoretical predicted times for their algorithm on a Cray T3D. According to the efficiency results we can conclude that our algorithm presents a significant better performance.

5   Conclusions

In this paper, we have propose a parallelization of the recursive decoupling method for solving tridiagonal linear system on distributed memory computer. The method showed an optimization of the memory requirements, a superlinear speedup and scalability. The memory savings come from a compressed storage policy which eliminates the null elements. On the other hand, we study the fill-in

in the algorithm to optimize the execution of the scalar algorithm. This way, the performance increases more than 91% for any value of N,

References

1. Amor, M., López, J., Argüello, F., Zapata, E. L.: Mapping Tridiagonal System Algorithms onto Mesh Connected Computers. International Journal of High Speed Computing **9** (1997) 101–126
2. Amodio, P., Brugnano, L.: The Parallel QR Factorization Algorithm for Tridiagonal Linear System. Parallel Computing **21** (1995) 1097–1110
3. Climent, J.-J., Tortosa, L., Zamora, A.: "A Recursive Decoupling Method for solving Tridiagonal Linear System in a BSP Computer". Proceedings in X Jornadas de Paralelismo (1999) 73–78
4. Cox, C. L., Knisley, J. A.: A Tridiagonal System Solver for Distributed Memory Parallel Processors with Vector Nodes. Journal of Parallel and Distributed Computing **13** (1991) 325–331
5. Dodson, D. S., Levin, S. A.: A Tricyclic Tridiagonal Equation Solver. SIAM J. Matrix Anal. Appl. **13** (1992) 1246–1254
6. Eğecioğlu, Ö., Koç, Ç. K., Laub, A.J.: A Recursive Doubling Algorithm for Solution of Tridiagonal System on Hypercube Multiprocessor. J. of Computational and Applied Mathematics **27** (1985) 95–108
7. Golub, G. H., Van Loan, C. F.: Matrix Computations. The Johns Hopkins University Press (1989)
8. Groen, P. P. N. de: Base–p–Cyclic Reduction for Tridiagonal System of Equations. Applied Numerical Mathematics **8** (1991) 117–125
9. Hockney, R. W., Jesshope, C. R.: Parallel Computers. Adam Hilger (1988)
10. Krechel, A., Plum, H.-J., Stüben, K.: Parallelization and Vectorization Aspects of the Solution of Tridiagonal Linear System. Parallel Computing **14** (1990) 31–49
11. Lin, F. C., Chung, K. L.: A Cost-Optimal Parallel Tridiagonal solver". Parallel Computing **15** (1990) 189-199.
12. Lin, W.-Y., Chen, C.-L.: A Parallel Algorithm for Solving Tridiagonal Linear Systems on Distributed–Memory Multiprocessors. International Journal of High Speed Computing, **6** (1994) 375–386
13. Ishihata, H., Takahashi, M., Sato, H.: Hardware of AP3000 Scalar Parallel Server. FUJITSU Sci. Tech. J. **33(1)** (1997) 24–30
14. Mattor, N., Williams, T. J., Hewett, D. W.: Algorithm for Solving Tridiagonal Matrix Problems in Parallel. Parallel Computing **21** (1995) 1769–1782
15. Müller, S. M., Scheerer, D.: A Method to Parallelixe Tridiagonal Solvers. Parallel Computing, **17** (1991) 181–188
16. Spaletta, G., Evans, D. J.: The Parallel Recursive Decoupling Algorithm for Solving Tridiagonal Linear Systems. Parallel Computing. **19** (1993) 563–576
17. Wang, X., Mou, Z. G.: The Parallel Recursive Decoupling Algorithm for Solving Tridiagonal Linear Systems. Proceedings of the third IEEE Symposium of Parallel and Distributed Processing (1991) 810–817

Fully Vectorized Solver for Linear Recurrence Systems with Constant Coefficients

Przemysław Stpiczyński[1] and Marcin Paprzycki[2]

[1] Department of Computer Science, Marie Curie-Sklodowska University, Plac Marii Curie-Sklodowskiej 1, 20–031 Lublin, Poland, phone: +4881 5376102, fax: +4881 5333669, e-mail: przem@golem.umcs.lublin.pl

[2] Scientific Computing Program, University of Southern Mississippi, Hattiesburg, MS 39406-5106, USA, phone: 601-266-6639, fax: 601-266-6452, e-mail: marcin@orca.st.usm.edu

Abstract. We describe the use of BLAS kernels as a key to efficient vectorization of m-th order linear recurrence systems with constant coefficients. Applying the Hockney-Jesshope model of vector computation, we present the performance analysis of the algorithm which considers also the influence of memory bank conflicts. The theoretical analysis is supported by experimental data collected on two Cray vector computers.

Keywords. m-th order linear recurrence systems, BLAS, LAPACK, vectorization, memory bank conflicts, speedup.

Conference topics. Numerical methods, Parallel and distributed algorithms.

1 Introduction

The critical part of several numerical algorithms reduces to the solution of a linear recurrence system of order m for n equations with constant coefficients [13, 16]:

$$
x_k = \begin{cases} 0 & \text{for } k \leq 0 \\ f_k + \sum_{j=1}^{m} a_j x_{k-j} & \text{for } 1 \leq k \leq n. \end{cases} \tag{1}
$$

The efficient solution to this problem is of particular interest in case of vector computers as optimizing compilers are not able to generate machine code that would fully utilize the underlying hardware. As our experiments show, even Cray's Fortran compiler, usually recognized as the best vectorizing compiler on the market, is in this category (see Section 5). In addition, numerical libraries (like LAPACK [1], implemented in the Cray's **scilib** library) instead of problem (1) provide a solution to a more general problem:

$$
x_k = \begin{cases} 0 & \text{for } k \leq 0 \\ f_k + \sum_{j=k-m}^{k-1} a_{kj} x_j & \text{for } 1 \leq k \leq n. \end{cases} \tag{2}
$$

Solution to this problem requires more memory and, in the case of LAPACK routines, the computational efficiency is obtained primarily by solving it for

multiple right hand sides. In case when the original problem (1) is solved, a simple application of a LAPACK routine does not result in achieving maximum performance (see Section 5). The aim of our work is thus to find the performance-optimal solver for the original problem (1). Based on our earlier work [9, 10, 11, 14] we have decided to approach the problem by augmenting the *divide-and-conquer* approach proposed there by application of BLAS kernels. We then proceeded to establish the optimal parameters to obtain maximum efficiency and to eliminate memory bank conflicts.

We proceed as follows. In the next section we introduce the algorithmic framework used in our work. We follow with the description of implementation details of the proposed algorithm. We then sketch the theoretical analysis of computational complexity. We complete our report by describing and analyzing results of our experiments performed on Crays C-90 and SV-1.

2 Algorithm description

In our considerations we will assume that $n \gg m$, i.e. the order of a recurrence system is rather small. The idea of the algorithm is to rewrite (1) as the following block system of linear equations

$$\begin{pmatrix} L & & & \\ U & L & & \\ & \ddots & \ddots & \\ & & U & L \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_p \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \vdots \\ \mathbf{f}_p \end{pmatrix}, \tag{3}$$

where for $q = n/p > m$ we have

$$L = \begin{pmatrix} 1 & & & & \\ -a_1 & \ddots & & & \\ \vdots & \ddots & \ddots & & \\ -a_m & & \ddots & \ddots & \\ & \ddots & & \ddots & \ddots \\ & & -a_m & \cdots & -a_1 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} -a_m & \cdots & -a_1 \\ & \ddots & \vdots \\ & & -a_m \\ 0 & & \end{pmatrix} \in \mathbb{R}^{q \times q}. \tag{4}$$

Note that L is a Toeplitz matrix, what means that entries are constant along each diagonal. The system (3) corresponds to the following recurrence system

$$\begin{cases} \mathbf{x}_1 = L^{-1}\mathbf{f}_1 \\ \mathbf{x}_j = L^{-1}\mathbf{f}_j - L^{-1}U\mathbf{x}_{j-1} \text{ for } j = 2,\ldots,p. \end{cases} \tag{5}$$

To solve this system let us consider the structure of the matrix

$$U = -\sum_{k=1}^{m}\sum_{l=k}^{m} a_{m+k-l}\mathbf{e}_k\mathbf{e}_{q-m+l}^T, \tag{6}$$

where $\mathbf{e}_k$ denotes k-th unit vector of $\mathbb{R}^q$. Obviously, equation (5) reduces to the form

$$\begin{cases} \mathbf{x}_1 = L^{-1}\mathbf{f}_1 \\ \mathbf{x}_j = L^{-1}\mathbf{f}_j + \sum_{k=1}^{m} \alpha_j^k \mathbf{y}_k \text{ for } j = 2,\ldots,p \end{cases} \tag{7}$$

where $L\mathbf{y}_k = \mathbf{e}_k$ and $\alpha_j^k = \sum_{l=k}^{m} a_{m+k-l} x_{(j-1)q-m+l}$. Note that to compute vectors $\mathbf{y}_k$ we need to find only the solution of the system $L\mathbf{y}_1 = \mathbf{e}_1$, namely $\mathbf{y}_1 = (1, y_2, \ldots, y_q)^T$. We can now form vectors $\mathbf{y}_k$ as follows

$$\mathbf{y}_k = (\underbrace{0,\ldots,0}_{k-1}, 1, y_2, \ldots, y_{q-k+1})^T. \tag{8}$$

This yields that the number of subsystems we must solve does not depend on the order of the system. To find vectors $\mathbf{z}_j$ and $\mathbf{y}_1$ we must solve $p+1$ recurrence systems of order m for q equations.

3 Implementation details

Now let us consider the possible implementations of the proposed algorithm. We can omit the assumption that $n = pq$ because after we choose integers p and q we can apply (7) to find $x_1, \ldots, x_{pq}$ and (1) to find $x_{pq+1}, \ldots, x_n$. First we have to find vectors $\mathbf{z}_j$ and $\mathbf{y}_1$. We can do it efficiently by using a sequence of _AXPY operations $\mathbf{y} \leftarrow \mathbf{y} + \alpha\mathbf{x}$. Note that _AXPY consists of $2N$ floating point operations and it can be computed in a simple loop of length N. So let us define matrices

$$Z = (\mathbf{z}_1, \ldots, \mathbf{z}_p, \mathbf{y}_1), \ F = (\mathbf{f}_1, \ldots, \mathbf{f}_p, \mathbf{e}_1) \in \mathbb{R}^{q \times (p+1)}$$

and denote $Z_{k,*}$ as a k-th row of Z. Now we can find the solution of the system $LZ = F$ using the formula

$$Z_{k,*} = \begin{cases} 0 & \text{for } k \leq 0 \\ F_{k,*} + \sum_{j=1}^{m} a_j Z_{k-j,*} & \text{for } 1 \leq k \leq q. \end{cases} \tag{9}$$

Initially columns of the matrix F can be stored in a one-dimensional array $\mathbf{x}$, so Z can be computed using the following code

```
do k=1,q
   do j=1,min(m,k-1)
      call saxpy(p+1,a(j),x(k-j),q,x(k),q)
   end do
end do
```

It can be easily calculated that the number of _AXPY operations is equal to $m(q - \frac{m+1}{2})$ and thus the total number of operations needed to find vectors $\mathbf{z}_j$ and $\mathbf{y}$ can be expressed as

$$C_1 = 2(p+1)m(q - \frac{m+1}{2}). \tag{10}$$

As soon as the matrix Z is calculated its last column ought to be copied to a new array y such that y(-m:0)=0.0.

```
call scopy(q,x(p*q+1),1,y(1),1)
do j=-m,0
  y(j)=0.0
end do
```

Now vectors $\mathbf{x}_j$, $j = 2, \ldots, p$, can be computed. For each vector we should compute coefficients α_j^k using the following code

```
do k=1,m
  call saxpy(m+1-k,a(m+1-k),x(q*(j-1)-m+k),1,alpha,1)
end do
```

and then find $\mathbf{x}_j$ using a sequence of _AXPY calls

```
do k=1,m
  call saxpy(q,alpha(k),y(2-k),1,x(q*(j-1)+1),1)
end do
```

The total number of floating-point operations in this part of the algorithm is

$$C_2 = 2(p-1)\left(\sum_{k=1}^{m}(m+1-k)+mq\right). \tag{11}$$

Now let us consider possible modifications of the proposed algorithm. First, observe that the last step of the algorithm can be implemented in terms of level 2 BLAS using one call of _GEMV. More precisely, when we form

$$W = (\mathbf{y}_1, \ldots, \mathbf{y}_m) \in \mathbb{R}^{q \times m} \tag{12}$$

then instead of the last loop above, we can use

```
call sgemv('N',q,m,1,w,ldw,alpha,1,1,x(q*(j-1)+1),1)
```

Note that the use of _GEMV requires additional space for qm entries of W.

Let us now observe that for finding Z we can consider the use of the routine _TBTRS from the LAPACK library [1] which solves a system $AX = B$ where A is a triangular banded matrix. Thus instead of the sequence of _AXPY calls based on (9) we would have the following LAPACK call

```
call stbtrs('L','N','U',q,m,p+1,ab,ldab,x,q,info)
```

We have to recall that this routine does not take into account the Toeplitz structure of the matrix L and requires additional space for $m + 1$ diagonals of L, i.e. for $(m + 1)q$ additional values.

In the table below we summarize algorithms that can be used to solve the original problem (1):

Algorithm	Description
Scalar	Scalar code based on a direct implementation of (1)
Algorithm 1A	The main algorithm based on calls to the _AXPY routine
Algorithm 1B	As Algorithm 1A but the last step is calculated by one call of the level 2 BLAS routine _GEMV
Algorithm 2	The system $LZ = F$ solved by a call to the LAPACK routine _TBTRS and the last step calculated by the call to _GEMV
Algorithm 3	LAPACK _TBTRS routine called for one RHS

4 Performance analysis

To study the performance of the algorithm let us consider the theoretical model of vector computations introduced by Hockney and Jesshope [6, 2].

The performance r_N of a loop of length N can be expressed in terms of two parameters r_∞ and $n_{1/2}$ which are specific for a kind of loop and vector computer. The first parameter represents the performance in Mflops for a very long loop, while the second the loop length for which a performance of about $r_\infty/2$ is achieved. Then

$$r_N = \frac{r_\infty}{n_{1/2}/N + 1} \text{ Mflops.} \tag{13}$$

This yields that the execution time of _AXPY is

$$T_{AXPY}(N) = \frac{2N}{10^6 r_N} = \frac{2 \cdot 10^{-6}}{r_\infty}(n_{1/2} + N) \text{ seconds.} \tag{14}$$

From (10), (11) and (14) we get that the total execution time of our algorithm can be estimated as follows

$$T(p,q) = \frac{2 \cdot 10^{-6}}{r_\infty} m \left(2pq + 2n_{1/2}p + n_{1/2}q - 2.5n_{1/2} - 0.5n_{1/2}m - m - 1\right),$$

where $n = pq$. It can be easily verified that $T(p,q)$ reaches its minimum at the point

$$(p,q) = (\sqrt{n/2}, \sqrt{2n}). \tag{15}$$

Thus the optimal choice of p and q depends only on the problem size n and because these numbers should be integers we choose $q = \lfloor\sqrt{2n}\rfloor$ and $p = \lfloor n/q\rfloor$. Here, the last $n - pq$ elements of the solution $\mathbf{x}$ can be computed by a scalar algorithm based on (1).

Sometimes these chosen parameters have to be adjusted to avoid memory bank conflicts. Vector computers usually store data so that contiguous words (e.g. elements of arrays) are in separate memory banks. Usually the number of banks in the memory system is a power of two. Memory bank conflicts may occur when an array's stride (the difference in the index between two successive iterations) is a multiple of a power of two. Then the memory cannot be efficiently

used because CPU must wait until a former memory request to the same bank is completed. Thus to avoid memory bank conflicts the parameter q should be chosen as follows

$$q = \begin{cases} \lfloor \sqrt{2n} \rfloor - 1 & \text{if } \lfloor \sqrt{2n} \rfloor \text{ even,} \\ \lfloor \sqrt{2n} \rfloor & \text{otherwise.} \end{cases} \quad (16)$$

Finally let us calculate the number of floating point operations performed by the algorithm. Adding C_1 and C_2 defined by (10) and (11), and the number of flops required for finding the last $n - pq$ entries of the solution we get

$$C_{n,m}(p,q) = C_1 + C_2 + m(n - pq - \frac{m+1}{2}) = 3mpq - \frac{5}{2}m(m+1) + mn. \quad (17)$$

5   Results of experiments

The method has been implemented in FORTRAN and tested on a single processor of the Cray C-90 and SV-1 vector computers. We have used the optimized versions of BLAS and LAPACK available in the **scilib** library. Each algorithm was tested varying the problem sizes n and m and values of parameter q. CPU time was measured using the *second* function and the presented results represent the best values from multiple runs.

Figures 1 and 2 illustrate the dependency between the performance of Algorithm 1A and the value of parameter q for $m = 1$ and $n = 64000$ and $n = 1024000$ respectively. Results for both Cray's are reported in $Mflops$.
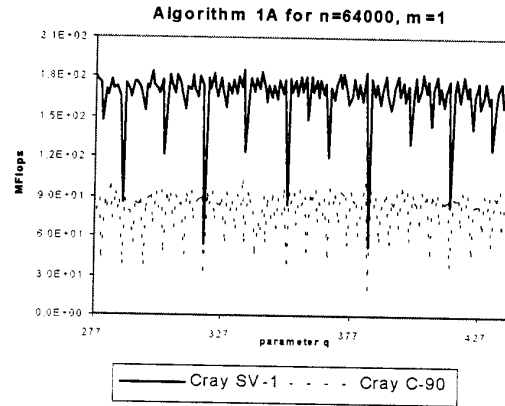

Fig. 1. Performance of Algorithm 1A for various values of q
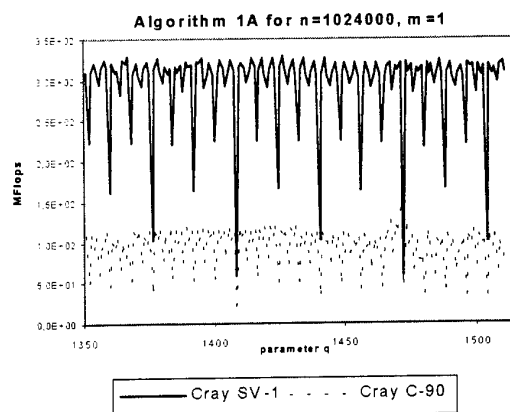
Algorithm 1A for n=1024000, m=1

Fig. 2. Performance of Algorithm 1A for various values of q

It was shown above (see Section 4) that the optimal value of the parameter q depends only on the size of the problem n. Our experiments support this claim and show that this result holds for both machines (the optimal value is the same on both Crays) even though they have different characteristic parameters r_∞ and $n_{1/2}$. The experimental optimal value of q has been found to be in close proximity of the theoretically predicted one (excluding values which are powers of 2 for which the memory bank conflicts affect performance). Thus, in computational practice, the theoretically predicted optimal value of q can be used to implement the code.

Figures 3 and 4 depict the relationship between the performance (in *Mflops*) and the size of the problem n and the order of the recurrence m (for these experiments the theoretically predicted optimal value of q was used). In Figure 3 we report the results for $n = 64000$ and $m = 1, 2, \ldots, 6$ for both Crays and all five algorithms. In Figure 4 we present similar results for $n = 1024000$.

First, let us observe that the qualitative behavior of the five algorithms is the same for both machines and is independent of the problem size n.

For $m = 1$ the Algorithm 1A is the most efficient. For Algorithms 2 and 3 a performance dip manifests itself for $m = 2$. Starting from $m = 2$ further increase in m results in the performance increase. Interestingly, for all values of m, Algorithms 2 and 3 which utilize LAPACK library routine _TBTRS are substantially less efficient than Algorithms 1A and 1B and only barely more efficient than the Scalar code.

As m increases, Algorithm 1B outperforms Algorithm 1A. This can be explained as an effect of the application level 2 BLAS matrix-vector multiplication _GEMV.

Finally, note that the performance of the two Crays depends on the problem
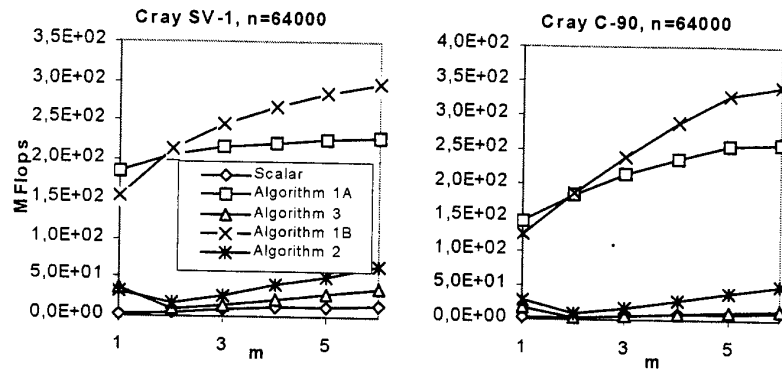
Fig. 3. Performance of the algorithms for various m
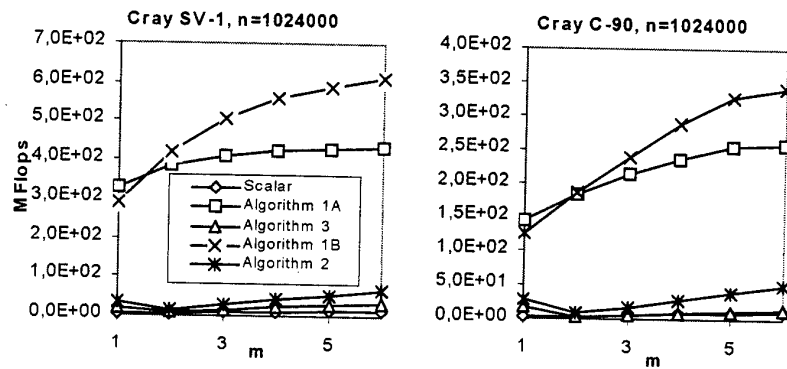
Fig. 4. Performance of the algorithms for various m

size (n). For small n Cray C-90 matches the performance of the newer SV-1 (for $m = 6$ it even outperforms it slightly). The situation changes radically for $n = 1024K$. Here, the Cray SV-1 is almost twice as fast as the Cray C-90.

We believe that from the point of view of the user one of the more interesting parameters is the speedup of the "fancy" algorithms over the basic Scalar approach. We illustrate this aspect of the problem in Figures 5 and 6. Here we report the speedup as the function of the problem size n for both machines for $m = 1$ and $m = 4$ respectively. As previously, the optimal theoretical value q was used for algorithms 1A, 1B and 2.
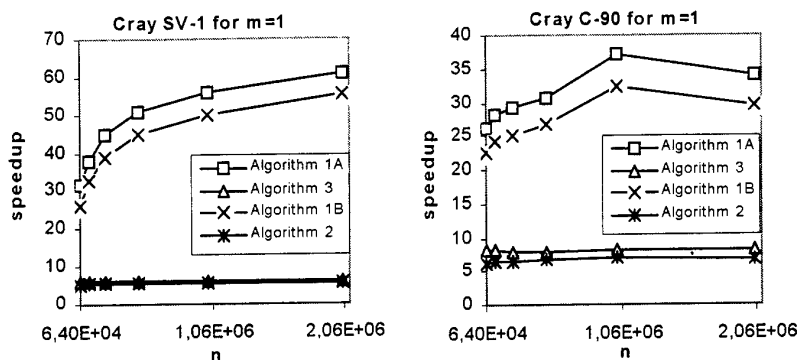
Fig. 5. Speedup of the algorithms for various n
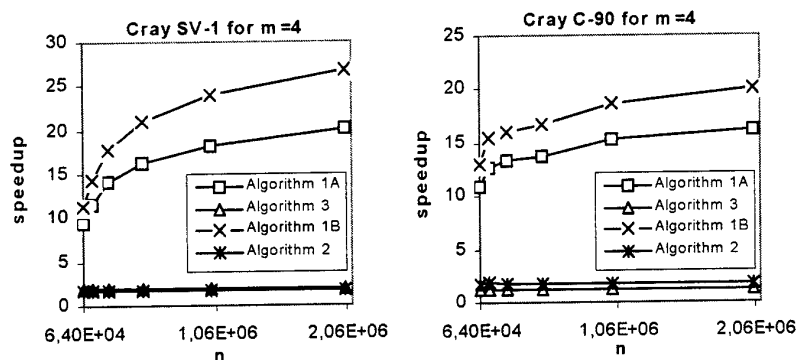
Fig. 6. Speedup of the algorithms for various n

As previously, the results are qualitatively similar for both machines. In all cases (independently of n) Algorithms 2 and 3 do not result in a significant speedup over the Scalar approach. Interestingly, while as n increases (for a fixed m, speedup of Algorithms 1A and 1B over Scalar increases, as m increases (for a given m) the speedup decreases. This indicates that the code generated by the compiler from the Scalar algorithm for increasing m results in improved efficiency.

Finally let us summarize the results of experiments

— Algorithms 1A and 1B achieve the best performance for values of the param-

eter q close to the theoretical optimal value. The optimal choice of q depends only on the problem size (and memory bank conflicts).

- The use of Algorithm 1B instead of 1A is profitable when $m > 2$. This is caused by the use of the level 2 BLAS routine _GEMV. However, use of _GEMV requires additional space for qm entries of W.
- The speedup of Algorithms 1A and 1B over the Scalar code increases when the problem size n increases and decreases when the order of the system m increases.
- The MFlop performance increases when the problem size n increases as well as when the order of the system m increases.
- When $q = a2^k$ (for integer a, k), performance rapidly decreases. Increase in the value of k results in further substantial performance degradation. This is the effect of memory bank conflicts.
- The performance of Algorithm 2 and 3 is rather poor and the algorithms require additional space. This is a result of the fact that the _TBTRS routine from LAPACK solves more general problem (2) and does not utilize the special Toeplitz structure of the matrix L.
- For first order linear recurrences ($m = 1$) Algorithm 1A is approximately six times faster then the Algorithms 2 and 3 which use _TBTRS routine from LAPACK and for large n achieves speedup up to 60 against the Scalar algorithm based on (1).

6 Acknowledgments

References

1. E. Anderson, Z. Bai, C. H. Bischof, J. W. Demmel, J. J. Dongarra, J. J. Du Croz, A. Greenbaum, S. J. Hammarling, A. McKenney, S. Ostruchov and D. C. Sorensen, *LAPACK User's Guide* (SIAM, Pfiladelphia, 1992).
2. J. Dongarra, I. Duff, D. Sorensen and H. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers* (SIAM, Pfiladelphia, 1991).
3. J. Dongarra and L. Johnsson, Solving Banded Systems on Parallel Processor. *Parallel Comput.* 5(1987) 219–246.
4. D. Heller, A survey of parallel algorithms in numerical linear algebra. *SIAM Review* 20(1978) 740–777.
5. A.C. Greenberg, R.E.Lander, M.S. Paterson and Z. Galil, Efficient parallel algorithms for linear recurrence computation, *Inf. Proc. Letters* 15(1982) 31–35.
6. R. Hockney and C. Jesshope, *Parallel Computers: Architecture, Programming and Algorithms* (Adam Hilger Ltd., Bristol, 1981).
7. D.J. Kuck, *Structure of Computers and Computations* (Wiley, New York, 1978).

8. J. Modi, *Parallel Algorithms and Matrix Computations* (Oxford University Press, Oxford, 1988).

9. M. Paprzycki and P. Stpiczyński, Solving linear recurrence systems on a Cray Y-MP, in: J.Dongarra, J. Waśniewski eds., Lecture Notes in Computer Science 879, (Springer–Verlag, Berlin, 1994) 416–424.

10. M. Paprzycki and P. Stpiczyński, Solving linear recurrence systems on parallel computers, *Proceedings of the Mardi Gras '94 Conference*, Baton Rouge, Feb. 10–12, 1994 (Nova Science Publishers, New York, 1995) 379–384.

11. M. Paprzycki and P. Stpiczyński, Parallel solution of linear recurrence systems. *Z. Angew. Math. Mech.* 76(1996) Suppl. 2, 5-8.

12. A.H. Sameh and R.P. Brent, Solving triangular systems on a parallel computer. *SIAM J. Numer. Anal.* 14(1977) 1101–1113.

13. J. Stoer and R. Bulirsh, *Introduction to Numerical Analysis* (Springer, New York, 2nd ed., 1993).

14. P. Stpiczyński, Parallel algorithms for solving linear recurrence systems, in: L. Bougé et al. eds., Lecture Notes in Computer Science 634, (Springer–Verlag, Berlin, 1992) 343–348.

15. P. Stpiczyński, Error analysis of two parallel algorithms for solving linear recurrence systems, *Parallel Comput.* 19(1993) 917–923.

16. P. Stpiczyński and M. Paprzycki, Parallel algorithms for finding trigonometric sums, in: D.H. Bailey et al. eds., *Parallel Processing for Scientific Computing*, (SIAM, Philadelphia, 1995) 291–292.

17. H. A. Van Der Vorst and K. Dekker, Vectorization of linear recurrence relations, *SIAM J. Sci. Stat. Comput.*, 16(1989) 27-35.

Parallel Solvers for Discrete-Time Periodic Riccati Equations*

Rafael Mayo[1], Enrique S. Quintana-Ortí[1], Enrique Arias[2], and Vicente Hernández[2]

[1] Depto. de Informática, Universidad Jaume I, 12080–Castellón, Spain; {mayo,quintana}@inf.uji.es. Tel.: +34-964-728000. Fax: +34-964-728435.
[2] Depto. de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 46071–Valencia, Spain; {earias,vhernand}@dsic.upv.es. Tel.: +34-96-3877350. Fax: +34-96-3877359.

Abstract. This paper analyzes the parallel performance of a numerical solver for discrete-time periodic Riccati equations. The approach performs a sequence of orthogonal reordering transformations of the monodromy matrices associated with the equations, and then employs the so-called matrix disk function to solve a series of discrete-time algebraic Riccati equations. The experimental results report the performance of the parallel algorithms on a cluster of Intel Pentium-II processors.

1   Introduction

Consider the discrete-time linear systems

$$
\begin{aligned}
x_{k+1} &= A_k x_k + B_k u_k, \quad x_0 = \tilde{x}, \\
y_k &= C_k x_k,
\end{aligned}
\tag{1}
$$

$k = 0, 1, \ldots$, where $A_k \in \mathbb{R}^{n \times n}$, $B_k \in \mathbb{R}^{n \times m}$, and $C_k \in \mathbb{R}^{r \times n}$. Discrete-time periodic systems satisfy $A_{k+p} = A_k$, $B_{k+p} = B_k$, $C_{k+p} = C_k$, for some integer period p. The analysis and design of these systems has received considerable attention in recent years (see, e.g., [7,9,19,20,23]).

An important application in control theory is the linear-quadratic optimal control problem. The solution of this problem is intrinsically related to the unique periodic symmetric positive semidefinite solution, $X_k = X_{k+p} \in \mathbb{R}^{n \times n}$, of the discrete-time periodic Riccati equation (DPRE)

$$
\begin{aligned}
0 = {}& C_k^T Q_k C_k - X_k + A_k^T X_{k+1} A_k \\
& - A_k^T X_{k+1} B_k (R_k + B_k^T X_{k+1} B_k)^{-1} B_k^T X_{k+1} A_k.
\end{aligned}
\tag{2}
$$

Here, $Q_k = Q_{k+p} \in \mathbb{R}^{r \times r}$ is a positive semidefinite matrix of weights for the outputs, and $R_k = R_{k+p} \in \mathbb{R}^{m \times m}$ is a positive definite matrix of weights for the inputs (see [7] for details). In case $p = 1$, the DPRE in (2) reduces to the

well-known discrete-time algebraic Riccati equation (DARE) [15]. Traditional DARE solvers are described, e.g., in [14–16,18].

Consider now the periodic symplectic matrix pencil, associated with the DPRE (2),

$$L_k - \lambda M_k = \begin{bmatrix} A_k & 0 \\ -C_k^T Q_k C_k & I_n \end{bmatrix} - \lambda \begin{bmatrix} I_n & B_k R_k^{-1} B_k^T \\ 0 & A_k^T \end{bmatrix} \equiv L_{k+p} - \lambda M_{k+p},$$
(3)

where I_n denotes the identity matrix of order n. In case A_k is invertible, it is possible to construct the periodic monodromy matrix [9],

$$\Pi_k = M_{k+p-1}^{-1} L_{k+p-1} \cdots M_k^{-1} L_k, \quad \Pi_k = \Pi_{k+p},$$
(4)

and the solution of the DPRE can then be obtained by a spectral division technique [10,13,17]. Unfortunately, this is not a practical approach as a considerable loss of accuracy can be expected in case any of the inverses in (4) is ill-conditioned [11].

The Schur vectors method [2,12,15] was successfully extended in [9,11] for solving DPRE, without explicitly forming the corresponding monodromy matrices. In this method, a periodic Schur form of the monodromy matrix is computed with a special ordering of the eigenvalues. However, the parallel implementation of this type of algorithms (e.g., the QR/QZ algorithms) renders a poor scalability and an efficiency far from those of traditional matrix factorizations such as, e.g., LU decomposition [8].

In this paper we follow a different approach, described in [5], for the solution of DPRE. The algorithm employs a reliable swapping of the matrix products in (4) to transform the DPRE to p DAREs. We then employ the matrix disk function to obtain the corresponding solutions [4].

In sections 2 and 3 we briefly review, respectively, the "swapping" method for solving DPRE and the matrix disk function for solving DARE. In section 4 we describe the parallel implementations of the algorithms. Our medium-grain parallel approach requires efficient parallel implementations of two numerical kernels provided, e.g., in ScaLAPACK [8]. In section 5 we report the performance of the parallel implementations on a cluster of Intel Pentium-II processors, connected via a Myrinet switch crossbar network. Our concluding remarks are presented in section 6.

2 The Swapping Method for the DPRE

In [5] an algorithm is described for solving of DPRE without explicitly forming the monodromy matrices. The approach relies on the use of the following lemma.

Lemma. Consider the matrix pair (Z, Y), $Z, Y \in \mathbb{R}^{n \times n}$. *If* Y *is invertible and*

$$\begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} Y \\ -Z \end{bmatrix} = \begin{bmatrix} R \\ 0 \end{bmatrix}$$
(5)

is a QR factorization of $[Y^T, -Z^T]^T$, *then* $Q_{22}^{-1}Q_{21} = ZY^{-1}$.

This lemma is applied in [5] to reorder the matrix products in (4). The goal is to obtain a matrix product of the form

$$\Pi_k = \hat{M}_k^{-1}\hat{L}_k, \tag{6}$$

without computing the inverses. The solutions of the DPRE are then computed by solving the corresponding DAREs.

Specifically, the method proceeds as follows. Consider $p = 3$, the monodromy matrix

$$\Pi_0 = M_2^{-1}L_2M_1^{-1}L_1M_0^{-1}L_0, \tag{7}$$

and apply the swapping to matrix pairs

$$(L_2, M_1), (L_1, M_0), \quad \text{and} \quad (L_0, M_2). \tag{8}$$

(Notice that the same matrix pairs also arise in Π_1 and Π_2.) Then, we obtain

$$(L_2^{(1)}, M_1^{(1)}), (L_1^{(1)}, M_0^{(1)}), \quad \text{and} \quad (L_0^{(1)}, M_2^{(1)}) \tag{9}$$

which satisfy

$$\begin{aligned} L_2M_1^{-1} &= (M_1^{(1)})^{-1}L_2^{(1)}, \\ L_1M_0^{-1} &= (M_0^{(1)})^{-1}L_1^{(1)}, \quad \text{and} \\ L_0M_2^{-1} &= (M_2^{(1)})^{-1}L_0^{(1)}. \end{aligned} \tag{10}$$

Therefore,

$$\Pi_0 = M_2^{-1}(M_1^{(1)})^{-1}L_2^{(1)}(M_0^{(1)})^{-1}L_1^{(1)}L_0, \tag{11}$$

and similar reorderings are obtained for Π_1 and Π_2. By repeating the swapping procedure with the matrix pair $(L_2^{(1)}, M_0^{(1)})$ we obtain $(L_2^{(2)}, M_0^{(2)})$ such that $L_2^{(1)}(M_0^{(1)})^{-1} = (M_0^{(2)})^{-1}L_2^{(2)}$ and the required reordering for Π_0 is obtained

$$\Pi_0 = \left(M_0^{(2)}M_1^{(1)}M_2\right)^{-1}\left(L_2^{(2)}L_1^{(1)}L_0\right) = \hat{M}_0^{-1}\hat{L}_0. \tag{12}$$

Similar reorderings are obtained for Π_1 and Π_2.

The algorithm can be stated as follows [5].

```
for  k = 0, 1, . . . , p − 1
     Set  L̂ₖ = Lₖ ,  M̂₍ₖ₊₁₎ mod p = Mₖ
end
for  t = 1, 2, . . . , p − 1
     for  k = 0, 1, . . . , p − 1
          (L₍ₖ₊ₜ₎ mod p, Mₖ) ← swap(L₍ₖ₊ₜ₎ mod p, Mₖ)
          L̂ₖ ← L₍ₖ₊ₜ₎ mod p L̂ₖ
          M̂₍ₖ₊ₜ₊₁₎ mod p ← Mₖ M̂₍ₖ₊ₜ₊₁₎ mod p
     end
end
```

The matrix products Π_k can still be formed in a formal way and reveal the monodromy relation if (some of) the A_k's are singular; see [6,22] for details. The computational cost of the reordering algorithm is $34p(p-1)n^3/3$ flops (floating-point arithemtic operations) and $\mathcal{O}(pn^2)$ for workspace.

3   The Matrix Disk Function for the DARE

In [13], Malyshev introduced an "inverse free iteration" for computing the right deflating subspace of a matrix pair. The method was refined and made truly inverse free in [3], and was further improved in [21].

Given a regular matrix pair $(A_0, B_0) = (A, B)$, with $A, B \in \mathbb{R}^{n \times n}$, the inverse-free iteration generates the sequence of matrix pairs

$$
\begin{aligned}
A_{k+1} &= Q_{21} A_k, \\
B_{k+1} &= Q_{22} B_k,
\end{aligned}
\tag{13}
$$

with

$$
\begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} B_k \\ -A_k \end{bmatrix} = \begin{bmatrix} R_k \\ 0 \end{bmatrix},
\tag{14}
$$

a QR factorization of $\left[B_k^T, -A_k^T \right]^T$.

In case this iterative scheme is applied with the initial $2n \times 2n$ matrix pair $(A_0, B_0) = (\hat{L}_k, \hat{M}_k)$, the solution X^* of the associated DARE can be obtained from the converged matrix $L_\infty = \lim_{k \to \infty} A_k$ as follows. Let

$$
L_\infty = \begin{bmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \end{bmatrix}
\tag{15}
$$

be an $n \times n$ partition of L_∞. Then, X^* is the solution of the full-rank linear least-squares problem

$$
\begin{bmatrix} L_{12} \\ L_{22} \end{bmatrix} X^* = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} ;
\tag{16}
$$

see [5] for details.

The cost of solving a DARE using the inverse free iteration for the matrix disk function is $13n^3$ flops per iteration of (13)–(14), $13n^3/3$ for the LLS problem in (16), and $\mathcal{O}(n^2)$ for workspace.

4   Parallel Algorithms

Two approaches are possible for parallelizing the previous DPRE solver on a parallel distributed-memory system. First, in case p is large compared to the number of nodes, a coarse-grain strategy can be employed. In this case each swapping of a pair of matrices (i.e., each QR factorization) is performed on a different node of the system, and each DARE is finally solved on a different node. The communications can be arranged so that a ring topology is sufficient (see [5] for details). This algorithm only requires tuned send/receive communication routines, an efficient numerical kernel for the QR factorization, like that, e.g., in [1], and a serial implementation of the inverse free iteration for the matrix disk function.

Nevertheless, in case the number of nodes, n_p, is larger than the period of the system (as can be expected in large multicomputers), in a coarse-grain algorithm part of the nodes of the system would be idle. Thus, in such case it is more efficient to perform each swapping in parallel. This medium-grain approach benefits from the existence of parallel linear algebra libraries, as ScaLAPACK [8], which implements, among others, a parallel kernel for the QR factorization. By sometimes performing the swapping algorithm on slightly larger matrices, of the form $[A^T, \ 0_{n \times k}^T, \ B^T]^T$, we avoid the redistribution of the matrices that would be necessary to combine different pairs of matrices. After the swapping stage, the DARE are solved using a parallel ScaLAPACK-like implementation of the matrix disk function.

5 Experimental Results

All the experiments were performed on a cluster of Intel Pentium-II processors connected via a *Myrinet* switch, using IEEE double precision floating-point arithmetic ($\epsilon \approx 2.2 \times 10^{-16}$). A BLAS implementation specially tuned for this architecture was employed. Performance experiments with routine DGEMM achieved 200 Mflops (millions of flops per second) on one processor.

Our first experiment reports the execution time the parallel DPRE solver, using n_p=4, 9, and 16 nodes. Specifically, in Figure 1(a)–(c) we show the execution time of the parallel implementation of the swapping method (routine PDGGSWP) for DPRE with periods p=2, 4, and 10. In Figure 1(d), we report the execution time of the DARE solver based on the inverse free iteration for the matrix disk function (routine PDGGDSK), required in the final stage of the algorithm.

Figure 2 analyzes the scalability of the parallel routines. For this purpose, we report the Mflops rate per node for PDGGSWP and PDGGDSK with $n/\sqrt{n_p}$ fixed at 450 and 750, respectively. The constant performance of the Mflops rate shows the high scalability of both algorithms.

6 Concluding Remarks

We have investigated the performance of a parallel numerical solver for discrete-time periodic Riccati equations. The algorithm performs a sequence of orthogonal reordering transformations of the monodromy matrices associated with the equations, and transforms the problem to a series of discrete-time algebraic Riccati equations, which are then solved by using the matrix disk function. Experimental results on a cluster of Intel Pentium-II processors report a high performance and scalability of our parallel algorithms.
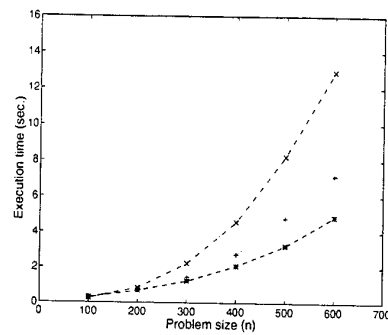
(a) PDGGSWP, $p = 2$
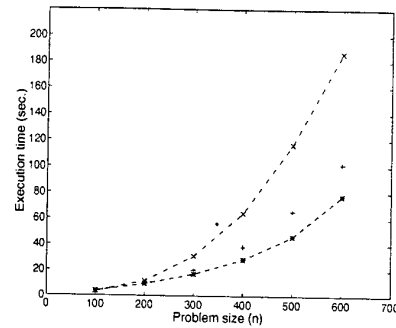
(b) PDGGSWP, $p = 6$
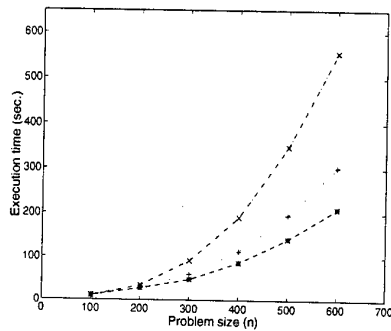
(c) PDGGSWP, $p = 10$

(d) PDGGDSK

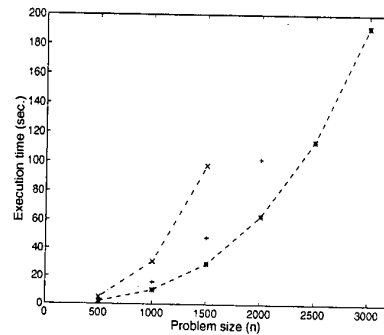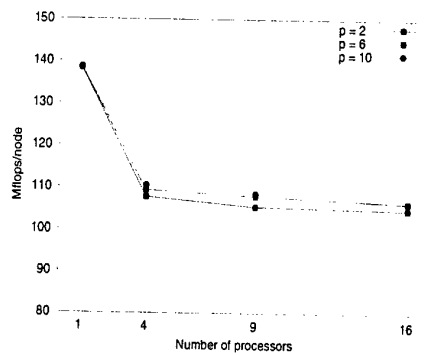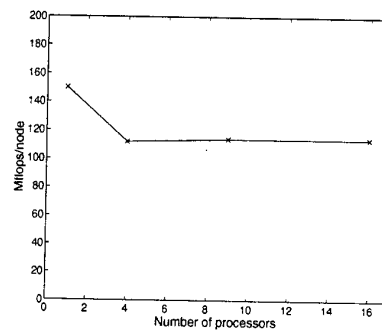**Fig. 1.** Execution times of the parallel routines.

(a) PDGGSWP, $n/\sqrt{n_p} = 450$

(b) PDGGDSK, $n/\sqrt{n_p} = 750$

Fig. 2. Scalability of the parallel routines.

References

1. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide.* SIAM, Philadelphia, PA, second edition, 1995.

2. W.F. Arnold, III and A.J. Laub. Generalized eigenproblem algorithms and software for algebraic Riccati equations. *Proc. IEEE*, 72:1746-1754, 1984.

3. Z. Bai, J. Demmel, and M. Gu. An inverse free parallel spectral divide and conquer algorithm for nonsymmetric eigenproblems. *Numer. Math.*, 76(3):279-308, 1997.

4. Z. Bai and Q. Qian. Inverse free parallel method for the numerical solution of algebraic Riccati equations. In J.G. Lewis, editor, *Proc. Fifth SIAM Conf. Appl. Lin. Alg., Snowbird, UT, June 1994*, pages 167-171. SIAM, Philadelphia, PA, 1994.

5. P. Benner. *Contributions to the Numerical Solution of Algebraic Riccati Equations and Related Eigenvalue Problems.* Logos-Verlag, Berlin, Germany, 1997. *Also:* Dissertation, Fakultät für Mathematik, TU Chemnitz-Zwickau, 1997.

6. P. Benner and R. Byers. An arithmetic for rectangular matrix pencils. In O. Gonzalez, editor, *Proc. 1999 IEEE Intl. Symp. CACSD, Kohala Coast-Island of Hawai'i, Hawai'i, USA, August 22-27, 1999* (CD-Rom), pages 75-80, 1999.

7. S. Bittanti, P. Colaneri, and G. De Nicolao. The periodic Riccati equation. In S. Bittanti, A.J. Laub, and J.C. Willems, editors, *The Riccati Equation*, pages 127-162. Springer-Verlag, Berlin, Heidelberg, Germany, 1991.

8. L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK Users' Guide.* SIAM, Philadelphia, PA, 1997.

9. A. Bojanczyk, G.H. Golub, and P. Van Dooren. The periodic Schur decomposition. algorithms and applications. In F.T. Luk, editor, *Advanced Signal Processing Algorithms, Architectures, and Implementations III*, volume 1770 of *Proc. SPIE*, pages 31-42, 1992.

10. G.H. Golub and C.F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, second edition, 1989.

11. J.J. Hench and A.J. Laub. Numerical solution of the discrete-time periodic Riccati equation. *IEEE Trans. Automat. Control*, 39:1197-1210, 1994.

12. A.J. Laub. A Schur method for solving algebraic Riccati equations. *IEEE Trans. Automat. Control*, AC-24:913-921, 1979.

13. A.N. Malyshev. Parallel algorithm for solving some spectral problems of linear algebra. *Linear Algebra Appl.*, 188/189:489-520, 1993.

14. V. Mehrmann. *The Autonomous Linear Quadratic Control Problem, Theory and Numerical Solution.* Number 163 in Lecture Notes in Control and Information Sciences. Springer-Verlag, Heidelberg, July 1991.

15. T. Pappas, A.J. Laub, and N.R. Sandell. On the numerical solution of the discrete-time algebraic Riccati equation. *IEEE Trans. Automat. Control*, AC-25:631-641, 1980.

16. P.H. Petkov, N.D. Christov, and M.M. Konstantinov. *Computational Methods for Linear Control Systems.* Prentice-Hall, Hertfordshire, UK, 1991.

17. J.D. Roberts. Linear model reduction and solution of the algebraic Riccati equation by use of the sign function. *Internat. J. Control*, 32:677-687, 1980. (Reprint of Technical Report No. TR-13, CUED/B-Control, Cambridge University, Engineering Department, 1971).

18. V. Sima. *Algorithms for Linear-Quadratic Optimization*, volume 200 of *Pure and Applied Mathematics*. Marcel Dekker, Inc., New York, NY, 1996.

19. J. Sreedhar and P. Van Dooren. Periodic descriptor systems: Solvability and conditionability. *IEEE Trans. Automat. Control*, 44(2):310–313, 1999.

20. J. Sreedhar and P. Van Dooren. Periodic Schur form and some matrix equations. In U. Helmke, R. Mennicken, and J. Saurer, editors, *Systems and Networks: Mathematical Theory and Applications, Proc. Intl. Symposium MTNS '93 held in Regensburg, Germany, August 2–6, 1993*, pages 339–362. Akademie Verlag, Berlin, FRG, 1994.

21. X. Sun and E.S. Quintana-Ortí. Spectral division methods for block generalized Schur decompositions. PRISM Working Note #32, 1996. Available from http://www-c.mcs.anl.gov/Projects/PRISM.

22. P. Van Dooren. Two point boundary value and periodic eigenvalue problems. In O. Gonzalez, editor, *Proc. 1999 IEEE Intl. Symp. CACSD, Kohala Coast-Island of Hawai'i, Hawai'i, USA, August 22–27, 1999* (CD-Rom), pages 58–63, 1999.

23. A. Varga. Periodic Lyapunov equations: some applications and new algorithms. *Internat. J. Control*, 67(1):69–87, 1997.

Thirty Years of Parallel Image Processing

Michael J B Duff

Department of Physics & Astronomy, University College London, Gower Street, London
WC1E 6BT, UK
MJBDuff@cs.com

Abstract. The history of the development of parallel computation methodology is closely linked with the development of techniques for the computer processing of images. In the early 60s, research in high energy particle physics began to generate extremely large numbers of particle track photographs to be analysed and attempts were made to devise automatic or semiautomatic systems to carry out the analysis. This stimulated the search for ways to build computers of increasingly higher performance since the size of the image data sets exceeded any which had previously been processed. At the same time, interest was growing in exploring the structure of the human visual system and it was felt intuitively that image processing computation should bear at least some resemblance to its human analogue.

This review paper traces the simultaneous progress in these two related lines of research and discusses how their interaction influenced the design of many parallel processing computers and their associated algorithms.

1. Thirty years ago

Image Processing was originally regarded as a subset of the wider field of Pattern Recognition which dealt with the analysis and processing of patterns in sound and other signal sources such as ECG and EEG as well as images. In all these areas, the research was mainly application driven. A three-day meeting in London in 1968, organised by the Institution of Electrical Engineers and entitled 'Conference on Pattern Recognition', comprised 37 papers. Of these, approximately one third were devoted to Optical Character Recognition (OCR) and a quarter to the physiology or psychology of human vision; the remainder was distributed more or less equally between studies of learning algorithms, speech recognition and general problems in pattern recognition. At this early stage, although it was realised that the principal application, OCR, would eventually demand much higher processing power than was currently available, the lack of effective algorithms meant that research was directed towards *how* to recognise images rather than to doing so at economic speeds.

Even so, what was not realised was how difficult the task would be. There was a quite unjustifiable optimism amongst researchers which could probably be excused by

the fact that everyone could observe in action (and, in fact, owned) a very effective image processing system which was portable, low power, high resolution and able to work in an unconstrained environment. Colour analysis, stereoscopy, time sequence analysis, automatic compensation for high or low light conditions, rotation invariance, image fragment recognition, learning capability: the system coped with all these difficult aspects. Unfortunately, it was considered that a combination of intuition and introspection would somehow reveal how the human vision system was constructed and that this knowledge could then be translated into an appropriate combination of hardware and software. This would amount to more than a PhD project but certainly should not take as long as ten years to complete.

In this optimistic atmosphere, there were two factors which stimulated an interest in faster computation. First, it seemed likely that useful algorithms would soon be developed and that computers would then need to be made much more powerful in order to achieve acceptable processing rates. Second, the progress being made in designing algorithms was poor, at least in part due to the inefficient computing services currently available. For example, at University College London in the early 60s, a large mainframe machine (IBM 360) provided the central computing service. Programs and even test images were entered via punched cards and then batch processed. Typically, a print-out of the results, using overprinted characters to represent image intensities, would be obtained on the following day; any small programming error (such as an unwanted comma) added a further day's delay to the program development time. In this virtually non-interactive environment, thinking constructively about algorithm design was almost painful.

Optimistic or not, almost all who were engaged in image processing research agreed that faster computers would, sooner or later, need to be developed and that there would be an immediate advantage if computing speeds could be improved. The important question was: how could a speed gain be achieved?

2. Faster computing

From the outset, it was clear that there were only three ways to speed up computing. They were (and still are):

a) More efficient programming;
b) Use of faster components;
c) Improved system hardware architecture.

With large data sets to be processed, it is extremely important to optimise the pieces of code in the so-called *inner loops*. For example, if the intensity of every pixel in an image is to be averaged with its neighbours, then the code performing the averaging may be executed a million times in a typical size image. Any wasted operations in that section of code will severely affect the overall efficiency of the program. It goes without saying that experienced programmers would not be expected to make this sort of error. In general, it would be hoped that most of the gains which could be obtained by efficient programming would normally already have been made.

Speeding up computers by using faster components is a continuous process of technological development which is largely under the control of computer manufacturers. In the period we are discussing, computing component technology moved from thermionic valves, through transistors to integrated circuits, having already progressed from mechanical (gear wheels) and electromechanical (relay) computation. In the last phase, integrated circuits have also undergone massive improvements in level of integration (numbers of components per unit area) and semiconductor technology, both of which have produced enormous speed gains. For the typical researcher, access to the best available circuit components has usually been a matter of cost since all new devices tend to be prohibitively expensive when first introduced.

The third approach is to redesign the computer architecture. The underlying structure of all computers was once much the same: there was a store for instructions, a store for data and a processor which was controlled by instructions extracted from the program store. These acted on data from the data store, producing a result which was returned to the data store. There were also units which input and output data and programs. A master controller ensured that all these operations were correctly sequenced. This extreme oversimplification hides all the ingenuity which went into making these basic operations efficient and transparent to the programmer.

Starting with this fundamentally simple architecture, the challenge was to make changes which would improve performance not marginally but substantially, ideally by many orders of magnitude. This was the impetus behind the introduction of Parallel Processing.

3. The Concept of Parallel Processing

Many hands make light work is a well known saying, but then so is *Too many cooks spoil the broth*. The fact is that increasing the size of the work force does not necessarily reduce the time (or cost) for completing a task. The introduction of additional labour implies a degree of organisation and co-ordination and may also require the task to be split up into manageable portions. The overhead for organisation can be more than the time saved and the task may not respond well to division. How often does one hear the comment: "I don't think you can help me; it will be quicker if I do it myself!"?

The central challenge in the design of parallel computers is to assemble many computers (or processors) into a system which will then share the execution of a program in such a way that the time between the start and end of the whole process is reduced. Ideally, if N computers are used to execute a program then the execution time T_N should be $(1/N)T_1$, where T_1 is the time taken by a single computer to execute the same program (suitably rewritten for a single computer). In practice, this ideal is seldom achieved, the exception being in computers designed for specific algorithms. A crude measure of efficiency of a parallel architecture is $T_1/(NT_N)$, but, as will be discussed in more detail later, this measure will depend on the program being

executed, both in relation to the task being performed and to the skill of the programmer.

4. Classifying Parallel Architectures

In general, a parallel computer will consist of an assembly of simple computers, usually referred to as processing elements (PEs). Each PE may be extremely simple, perhaps only capable of processing single bit data, but might alternatively be complex, such as a PC. There will usually be memory assigned to each PE and an interconnection network, both for transmitting data between PEs and for supplying instructions to the PEs. Some systems operate under the control of one master computer whereas others assign partial or even total autonomy to each PE.

In the past three decades, much has been written about the many different architectures of parallel processing computers and many attempts have been made to devise a taxonomy for classifying the architectures (e.g., see [9]). The best known attempt was by M J Flynn [8] whose classification was based on whether the data stream was single or multiple and on whether the instruction stream was single or multiple. Of the four possible classes, the one that most aptly fitted a representative group of parallel processing computers (several of which were actually constructed) was the SIMD class: an array of simple PEs all simultaneously executing the same instruction (Single Instruction stream), but each operating on its own part of the data (Multiple Data stream). However, despite the fact that the paper describing this taxonomy has been quoted in the literature more than has any other on this topic, this division of parallel processors into four classes is so crude as to be virtually useless. Many parallel systems either do not fall convincingly into any of the classes or else equally well fall into more than one. Furthermore, the first class (Single Instruction stream, Single Data stream) refers to serial computing so can hardly be treated as part of the taxonomy.

It is therefore not unreasonable to ask why researchers persist in attempting to devise classification schemes. There are probably two main reasons:

Divide and conquer   Computer scientists (and others) have experienced great difficulty in understanding the underlying principles of parallel processing systems and it can be a help if the structure of each system is compared with one of several archetypes: a form of learning by analogy;

Establishing design objectives   Parallel computer designers need to be clear what their strategy will be when designing a new system. It can be a useful design discipline to encapsulate a strategy by naming and defining the broad principles governing each particular design.

For the remainder of this review, classification schemes will not be considered, especially as there is now little or no agreement as to which scheme should be adopted.

5. Parallel Processing Fundamentals

5.1 Three Level Processing

It is easy to state in imprecise terms what is required of any parallel processing system. It is a system which, by employing more than one processor, completes a data processing task faster than could be achieved by a single processor. In order to investigate parallel architectures, the following discussion will concentrate on the particular problems associated with image processing. Examining the problems in detail, certain significant factors begin to emerge:

Data type Image data usually consist of large regular arrays of square picture elements (pixels), each of which represents the local brightness and, possibly, colour of the image. Typically, each pixel is assigned a 1-bit integer (black and white so-called binary images), an 8-bit integer (grey-level images) or a 24-bit integer (colour images). An image of approximately domestic television resolution (512 x 512 pixels) comprises rather more than one quarter of a million pixels. Very many image processing operations involve replacing each pixel by a new pixel whose intensity is a function of the intensities in a defined neighbourhood, for example, the 3 x 3 pixel region surrounding each pixel. This implies that an image processing operation can involve over 2.5 million basic operations (each requiring fetching data from memory, computing a sum or product and then storing the result in memory). The need for fast processing is self evident.

Computation type It is clear that the highly repetitive nature of the elements of the image processing computation might offer potential for structuring a computer architecture so as to take advantage of the repetitiveness.

Unfortunately, this brief analysis of image processing greatly oversimplifies the situation. Conventionally, the complex task of image processing is divided into three stages or levels [23]:

a) Low level processing which is characterised by taking in one or more images, processing them and outputting one or more result images. In general, the dimensions of the input and output data arrays will be identical;

b) Intermediate level processing in which the input data will be one or more images (input from the low level processing stage) and the output data will be one or more dimensionally smaller data sets, such as lists of detected object features and global properties of the image (e.g. average intensity, histograms, contrast range).

c) High level processing which attempts to extract meaning from the intermediate level data with a view to describing and analysing the input image. The output data might be as small as a single word or sentence.

5.2 Processor Arrays

As was discussed earlier, many low level image processing tasks can be broken down into identical short sequences of basic operations, each centred on every pixel in the image. An image architecture closely matching the apparent requirements of this level would therefore be an array of very simple processors, each associated with a single pixel and each accessing data only from its own local memory or from the neighbouring set of pixels. The repetitive nature of the processes to be performed would permit broadcasting a sequence of instructions to each simple processor (PE), the instructions being then executed simultaneously by every PE. This is the classic SIMD architecture.
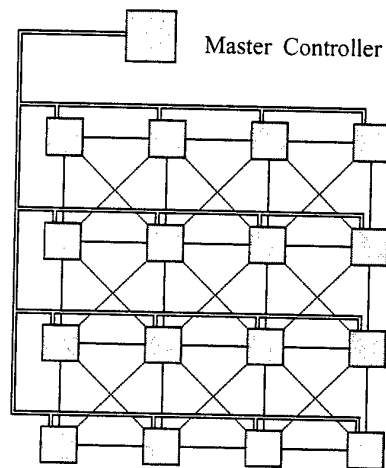

Fig. 1. A *4 x 4 PE array*, showing the *interconnections* between *PEs* and the *bus* distributing instructions in parallel to each *PE*

Apart from the paths taken by the instructions, all communication paths in the array are short (i.e. to nearest neighbours), provided that local memory is associated with every PE. One further set of longer paths is needed to input or output data to the memory array but these could be routed along the instruction highway. Fig. 1 illustrates the main features of a 4 x 4 PE array

Architectures of this type would appear to be ideal for low level processing but present many difficult problems in software design. Nevertheless, it can be shown that arrays of very simple PEs are theoretically capable of performing all image processing operations (even including those classified as intermediate or high level, although these might not be executed very efficiently).

One or more loosely coupled conventional processors can efficiently handle high level processing. There is no general pattern to the type of operations to be performed nor to the various types of input data set and the fastest available high speed workstation or even PC would usually offer the best solution. The same computer would probably be used to control the other two levels of the composite system.

The most difficult stage to implement is the intermediate level. By definition, the input data impose requirements similar to those for the low level but the need to abstract information derived from all parts of the image (or images) implies the need for efficient connection paths across the whole of the image array. It would also seem likely that an array of simple PEs would not represent an ideal structure for computing histograms and other results contained in comparatively small data sets. Optimisation is therefore difficult and likely to be specific task dependent.

A further problem resulting from the splitting of the low and intermediate levels is the difficulty in transferring the multiple image data between the two levels. Unless this can be achieved using many parallel paths, ideally one for each pixel, then this process might prove to be the bottleneck for the whole system.

Taking these two factors into consideration, there would seem to be good arguments for recombining the low and intermediate levels, enhancing the low level structure by adding good communication paths between all parts of the array of PEs.

In summary, the final assembly would comprise just two levels: the low/intermediate level would be an array of PEs, one per pixel for the size of image to be processed, and the high level/controller would be a conventional workstation or high performance PC.

5.3 Pipeline Processors

In the discussion in the previous section it was tacitly assumed that the task presented was to process a single image. Parallelism was achieved by assigning PEs to each part of the image data (i.e. to each pixel). An alternative approach can be adopted when many images are to be processed in a sequence. Under these circumstances, each processor is given a particular operation to perform and the sequence of images is fed through a string of processors, the output for the one providing the input for the next. The processors thus constitute a *pipeline* and the parallelism is now *function parallelism* rather than *data parallelism* (as was employed in the processor array). Sternberg has built and marketed several pipeline processors (named Cytocomputers) and developed complex software to program them [22].
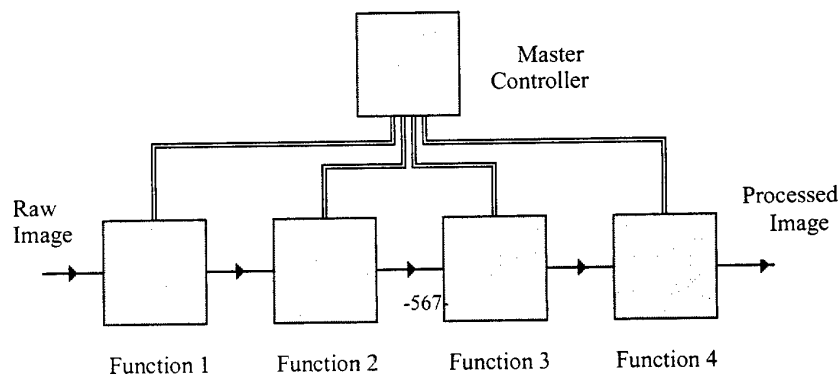
Fig. 2. A short pipeline processor with 4 *PEs* and a *master controller*

In passing, it is interesting to note that this type of computer might also be classified as SIMD in that each PE executes a single instruction on multiple data, although in this case the data is multiple in time rather than position. In that the Flynn system of classification appears not to distinguish between these two very different architectures, it would seem to be of little practical use.

Because the operations each PE performs on the image as it passes through it can be quite complex, a pipeline PE will usually be much more powerful than those utilised in processor arrays. A further consideration is that cost and program structure combine to make it unprofitable to construct very long pipelines; instead, it is more efficient to cycle each stream of images several times through the pipeline, reprogramming the PEs to perform new operations after each pass. Whether or not this is done, there is always the disadvantage that the so-called *latency* of the pipeline (the time delay between an image entering the first PE in the chain and the time it leaves the last PE) may be inconveniently long. For example, although a 100 PE pipeline might output fully processed images at a rate of 10 per second, the latency in the chain would be 10 seconds, thus ruling out such a system for real-time processing as might be required in a 'visually' controlled machine.

Other disadvantages are the difficulty in feeding forward partially processed images (to be used in combination later in the chain) and the virtual impossibility of handling feedback (when the parameters of the early stages of processing have to be adapted to the results of later stages).

5.4 MIMD Arrays

A third approach to parallel image processing makes use of a relatively small set of loosely coupled, powerful PEs, each capable of independent operation. A typical number would be 64 or less and the PE might be a microprocessor or even a PC. In principle, the image processing task is shared between all the PEs which then communicate over a high speed bus or some more complicated network. Each PE will have its own program store and substantial local memory whereas the system as a whole will usually be arranged so that one PE acts as a master controller and a major block of memory can be accessed by all the PEs. The classification Multiple Instruction stream, Multiple Data stream is clearly applicable since each PE executes its own program on its own part of the data.
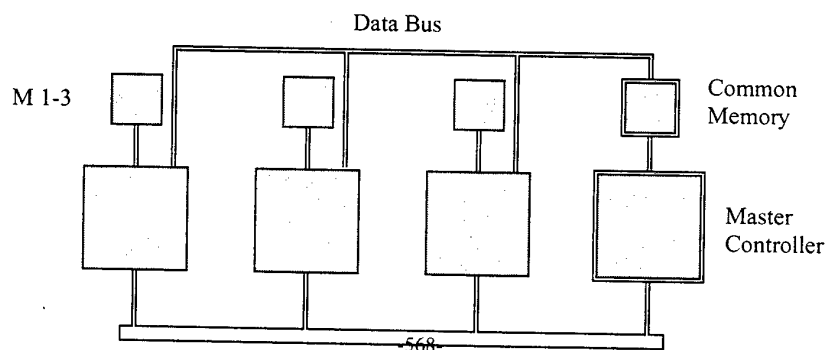
Fig. 3. Simple MIMD system with three PEs (each with local memory) and a master controller, together with a common memory block

MIMD systems have not made much impact on image processing. Just as employing more staff will not necessarily get a job done more quickly, so it has been found that PE 1-3 more PEs to an MIMD system does not always result in faster processing. Indeed, the additional overhead resulting both from subdividing the task and from communicating between the PEs can even result in a reduction of performance as more PEs are incorporated. The most serious objection to MIMD systems is that they are very difficult to program. Compilers which will efficiently segment the processing task into blocks, which will not leave PEs idle for much of the time, rarely exist and, in any case, : Instruction Bus :y over a range of different applications. It is therefore left to the programmer to decide how to employ the parallelism and this will imply that the programmer must know much more about the structure of the hardware system than is normal for software designers.

5.5 Special purpose devices

Faced with apparently insuperable difficulties in producing fast, efficient, general purpose image processing computers, some designers have tackled the more achievable challenge to design special purpose circuits which perform a very limited range of operations. For example, in some applications, an image transformed so that only the edges of objects are displayed (as white lines on a black background) can be useful. Another application needs to isolate only those parts of an image which are changing, perhaps because an object in the scene is moving.

Some of these devices combine a retina-like array of optical detectors with a matching array of hard-wired logic elements; other use a sequence of hard-wired processing units in a pipeline configuration. In today's jargon, systems such as these could be called *smart cameras* but their smartness is strictly limited and, somehow, disappointing.

5.6 Summary

There have been many approaches to parallelism in computers designed principally for image processing. The precise form of parallel architecture chosen is likely to depend on the range of tasks to be tackled. Thus, systems to be used for real-time control based on television cameras will almost certainly not be applicable to batch processing of large numbers of images collected by astronomers. Again, devices for

motion detection would have no place in a pathology laboratory dedicated to cervical smear analysis.

Parallel processing systems cannot be neatly categorised and it is doubtful whether there would be any value in doing so at this stage in their development. For those of us who have spent much of our working lives studying and designing such systems, it is discouraging to have to admit that the need for parallel systems in image processing has fallen to a low priority. The current obstacle to progress is the lack of effective algorithms; workstations and the latest generations of PC are usually quite fast enough for anything that needs to be done.

6. Historical Background

6.1 Pioneer research

Blindness is a terrible affliction. Most of the human environment is designed or has been adapted on the assumption that we can see and the vast majority of tasks performed by humans rely on human vision to provide the necessary feedback to control performance. Without the gift of vision, humans are greatly restricted in what they can do.

In the same way, the development of sophisticated automation, especially in the manufacturing industry, has been retarded by the lack of competent computer vision systems. This is particularly serious with respect to inspection of manufactured parts and similar problems occur in medicine in the areas of mass screening; the subject of optical character recognition has already been mentioned in this review. Pure science would also benefit if it were possible to automate the analysis of photographic images produced in many research areas, high energy particle physics and astronomy being the earliest of these to generate this requirement.

The inadequate performance of even the fastest available computers in the early 1960s (when the demand for computer vision was beginning to become apparent), stimulated computer scientists to turn their attention to the research that was then in progress investigating the mechanisms underlying biological vision. Two seminal papers in this area were the study of frog vision by Lettvin et al [14] and a slightly later paper by Hubel and Wiesel on cat vision [13]. Herscher and Kelley embodied the ideas behind the first paper in a hardware demonstration [12].

In the studies of both the frog and the cat, the anatomy of the visual system was seen to embody an array of photodetectors (rods and/or cones) forming the retina with the electrical outputs of the photodetectors being cross-connected, effecting both summation and lateral inhibition (i.e., a strong output from one photodetector reduces the strength of the output from its neighbours). The modified outputs are fed via a

bundle of nerve fibres (the optic nerve) through to the visual cortex of the brain where layer upon layer of densely interconnected neurons carry out parallel logic operations on the retinal outputs. In the case of the frog, only a very small number of image properties can be extracted from the optical data, such as detection of an object moving into the field of view. However, the cat's visual system is very similar to the human's and is therefore capable of greatly sophisticated scene analysis. In all these studies, the anatomical investigation was supplemented by physiological measurements and much was learnt about how the systems effected their processing.

At approximately the same time as this work was started, Unger published the first of his two papers [24],[25] proposing a processor array, although he did not build an array himself; in fact, these papers seem to be his last contact with the subject of computer architecture. His papers described a theoretical, square array of simple logic elements, each of which could receive data from or send data to any of its four neighbours. He demonstrated that his array could execute simple but useful functions on arrays of data of the same dimensions as the logic array but he did not suggest how these logic elements could be implemented in hardware. Fortunately, the Unger papers served to inspire others who then did construct hardware based on the ideas he had expressed. Another pioneer was Golay [10, 11] whose processor proposals, although conceived as a serial device, were turned into hardware by Preston [16] who was well aware that a more parallel version could have been constructed.

Computers whose designs were based loosely on Unger's ideas were, in order of construction, Solomon [19], ILLIAC III [17], ILLIAC IV [1],[21] and DAP [7]. It is not clear whether Solomon was actually constructed and made to operate but ILLIAC III caught fire before it could be completed and only 'worked' in simulation. ILLIAC IV was only partially completed but sufficient was built to enable it to carry out many large-scale computations. DAP started being developed in 1973, was prototyped in 1976 and put into commercial production in 1980. The last machines in this sequence were MPP [2] which first appeared in 1983 and the Connection Machine [13] which later evolved into the commercial CM series of massively parallel processor systems.

Parallel processing research in the Image Processing Group in the Department of Physics at University College London (UCL) was initially influenced by the biological papers listed above. The research into parallel processing followed some seven years of development of semi-automatic microscopes and other image analysis equipment (1958-1965), constructed for the three High Energy Particle Physics groups in the same department. Unger's paper was not seen by the UCL group until many years later and it was surprising to see how the two disconnected lines of research had by then converged. At this time, another field of research was also coming into being: Neural Networks. The pioneer work here was carried out by Rosenblatt [20] who devised the *Perceptron*. This circuit loosely simulated a neuron and introduced the idea of constructing circuits which could be trained to make decisions by adjusting the values of certain circuit elements (usually variable resistors) in response to a set of training patterns. The strengths of selected pattern features were translated into voltages which were then summed through the variable resistors (one for each feature). The resulting summed voltage was then compared with a threshold voltage and the pattern classified as class A (sum at or above the threshold) or class B (sum below the threshold). If necessary, the automatic trainer

then adjusted the variable resistors appropriately to correct the decision and a new pattern was presented. It could be shown that the process would converge so that, ultimately, all the circuit's decisions were correct for the training set and would generally be correct for similar but previously unseen patterns. This work was also influential on the UCL programme.

6.2 Research at UCL

UCPR1

A research grant application written in 1965 to request support for the UCL research programme is of interest. It could be submitted even today with very little modification since it addresses problems which are relevant to the design of parallel processing systems and are still unsolved:

> *'One of the main limitations on the design of neuron-like networks has been the prohibitive cost of constructing circuits which involve very large numbers of circuit elements together with a high degree of interconnection between the elements. If these limitations were to be removed by exploiting some of the relatively new techniques for the production of microminiature circuits, then it might prove possible to develop networks which would embody some of the considerable analytical facilities of neural nets. In addition, the increased component density would permit a measure of redundancy, and local failure would not impair efficiency of the net. This, in its turn, would allow the use of circuit construction techniques which do not produce component values within close tolerances.*

> *..........The last part of the proposed programme which has been envisaged would comprise:*

> *a) The construction of transistor models of neural elements with a view to producing a critical survey of their properties and to designing improved elements;*
> *b) Assembly of such elements into various arrays, exploring the numerous modes of interconnection;*
> *c) Simulation of such networks by means of computer programs, and developing appropriate mathematical methods to handle the logical circuit analysis;*
> *d) Translation of the circuitry into microminiature components, and utilisation of circuit replication techniques*
> *e) ....*

Application to the UK Department of Scientific and Industrial Research for support for a research programme entitled 'Pattern Recognition Matrices', dated March 1965'

This programme resulted in the construction and demonstration in September 1967 of UCPR1 [4]. Integrated circuits were not generally available at this time so the active components in UCPR1 were diodes and transistors. Regions of interest in photographs of the tracks of high energy charged particles (in nuclear emulsions and from cloud chambers and bubble chambers) are characterised by either a sharp change in direction or by a branching of the track into two or more components. Automated scanning equipment had been built which needed manual centring on these regions so UCPR1 was designed to show the possibility of making a retina-like device which would detect the regions automatically.
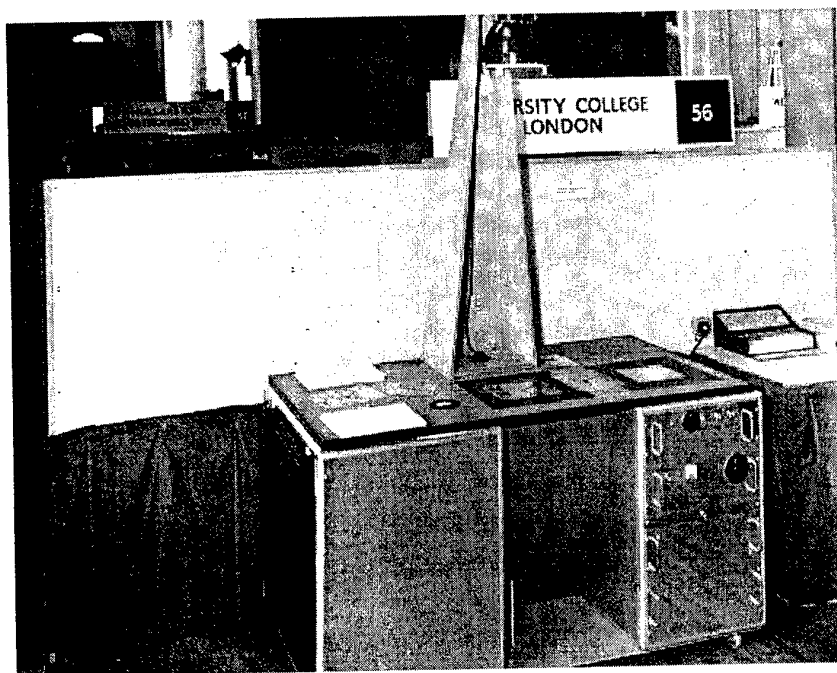

Fig. 4. A working demonstration of the parallel processor UCPR1, as demonstrated at the Physical Society Exhibition in London in 1967. The lamp at the top illuminates a track chamber photograph placed over an array of photodiodes. The electric lamp array to the right shows the location of vertices in the photograph.

The input to the system was a square array of 256 photodiodes onto which the track photograph was projected. Hard-wired circuits were layered under the photodiodes and performed the following functions:

1. Amplification;
2. Summation over a 3 x 3 window surrounding each photodiode;
3. Non-linear amplification of the summed output, saturated by at least two out of the nine possible inputs
4. Summation over the outer edge of the 5 x 5 window centred on each amplifier output
5. Comparison of the final output with a variable threshold, scanned from a high value downwards and designed to locate the maximum summed output.
6. The threshold scanner stopped scanning as soon as a maximum was detected and the final layer outputs were fed to a 256 x 256 array of light bulbs. The bulb or bulbs which lit indicated the position of the detected region of interest (referred to as a *vertex*). The variable threshold scanned at 50 Hz so vertices could be detected in real-time, i.e., once every 20msec.

The Diode Array

UCPR1 achieved what it set out to do: it successfully detected and located vertices in charged particle track photographs. A small piece of extra hardware showed that it could also be used to detect ends of lines and a further extension enabled UCPR1 to recognise carefully drawn alphanumerics (but not the complete alphabet) by analysing the locations of ends and vertices. The obvious weakness of the UCPR1 concept was that each layer of processors could only execute a single logic function. In effect, UCPR1 was unprogrammable.
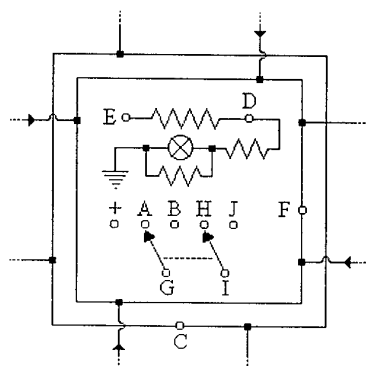
Fig. 5. A single PE of the Diode Array, showing a *neon indicator* (ON or OFF for one or zero outputs) and a *double-pole, double-throw switch* to allow zero or one to be entered

The Diode Array project [5] was the first attempt to determine what was the simplest specification for a processing element (PE) that would enable it to be programmed to perform all possible functions on arrays of data. Consideration of the experience gained in studying UCPR1 and also taking into account what was then known about the construction of the mammalian retina, led to the proposal that each PE should be able to input, store and output single-bit data, should be capable of inverting data and, finally, should be connected to neighbours in such a way that data from neighbours could be input as a logical OR of all four inputs.

The basic PE is shown in figure 5. It includes a neon bulb which glowed to show a 1 output (dark for a 0 output) and the points labelled A to J and + were initially left unconnected. The double pole switch was used to input a 1 or a zero (corresponding to its ON and OFF positions). A small 5 x 5 array was constructed and additional electromechanical relay circuits added to enable the user to systematically connect together various combinations of the labelled circuit points, the same combination in each PE. Treating the switch state as representing black and white image data, it could be demonstrated that functions such as image inversion, object edge extraction and object expansion and shrinking could be effected.

A computer simulation of the array was written together with a Monte Carlo program. This applied a wide range of random intra-processor connections (between the labelled points) with a view to discovering all differing image processing operations which could be implemented by the array. The otherwise exhaustive search was narrowed by eliminating obviously useless connections, such as connecting the positive voltage supply (+) to Earth. Rerunning the program many times established the existence of more than 70 processing functions. For reasons that are not clear, those functions which had been built into the hardware array were discovered by the Monte Carlo program earliest in its operation.

Because the connections between PEs were combined by OR-gates to provide a single input into neighbouring PEs, the array had no 'sense of direction'. For example, it would never be able to detect that one object lay above another in an image. Also, the obsolescent hardware components used to construct the array imposed undesirable constraints on the implementation of the logic functions. The next stage in the research programme utilised first small scale, next medium scale and finally large scale integration.

The CLIP Project

Continuing the search for the optimum PE design, a series of array processors was constructed. These so-called Cellular Logic Image Processors (CLIP1 to CLIP4) were gradually increased in complexity thus allowing each to be thoroughly understood before additional sophistication was permitted. CLIP1 and CLIP2 will not be described here as all their important features were included in CLIP3. CLIP3 will

also not be discussed in detail since its main purpose was to provide a design study for a fully integrated version which could be manufactured and marketed. In fact, for reasons of cost, CLIP4 was slightly less complex than CLIP3. The higher level of integration in CLIP4 (8 PEs per integrated circuit) made it economic to build an array of 96 x 96 PEs whereas CLIP3 had only 16 x 12 PEs and was not of practical value for applied image processing.

The logic functions of CLIP4 are shown in outline in Figure 6. At the heart of the PE are two identical minterm generators. Each has two binary data inputs (A, the value of the local pixel, and a composite value derived from another pixel value stored in B and from data from neighbours), one binary output and four binary control inputs. By applying any of the sixteen possible 4-bit binary control words to a generator, any of the sixteen possible Boolean combinations of the two inputs can be produced at the output. The output from the lower generator is distributed to neighbouring PEs and the upper output is stored as a result. Each PE, on receiving inputs from neighbours, selects a subset by means of a programmable gate and ORs the subset with a single bit stored in local memory (B in the figure). Further gates allow the PE to act as a full adder. Additional connections are used to input and output data to and from the array. The detailed operation of the PE is too complex to describe in the space available for this review but a full description of the CLIP3/CLIP4 systems can be found in [3],[6].
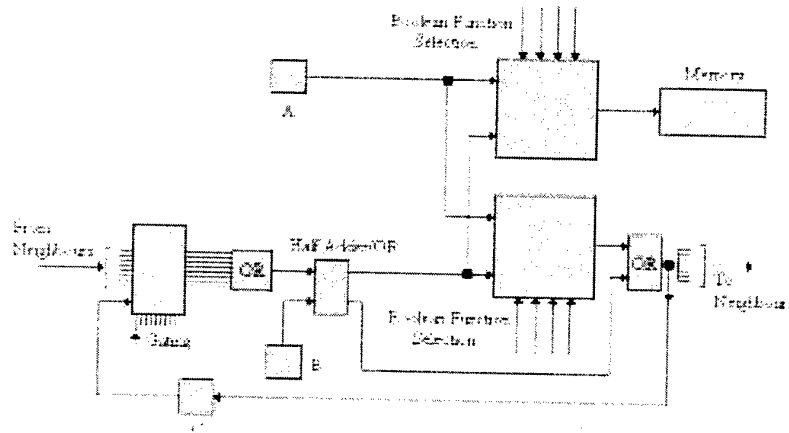

Fig. 6.Schematic logic diagram of the CLIP4 processing element

Three classes of operation can be performed by these CLIP processors. They are those in which:

- Each output pixel is a function only of the corresponding input pixel;
- Each output pixel is a function of the corresponding input pixel and of the eight pixels surrounding it;

- Each output pixel is a function of the corresponding input pixel and of any other connected to it by propagation through chains of neighbouring pixels.

One further feature of the array is an OR-gate (not shown in the figure) with inputs from every PE, used to determine whether a binary image stored in the PEs has at least one pixel which is non-zero. In general, the PE processes single bit binary data in each operation; multiple bit data is processed one bit at a time, i.e., bit-serially. Although beyond the scope of this review, it can easily be shown that an array of PEs with the features listed here can be programmed to perform all image operations and, indeed, all mathematical calculations. In short, CLIP3 and CLIP4 are universal computing systems.

The development of CLIP4 extended from 1974 to 1980. At that time, the CLIP4 integrated circuit was the largest ever to be manufactured in the UK under contract to the universities and the technical difficulties experienced were immense. After this worrying development period, CLIP4 was applied to many image processing projects and was in constant use for the next 10 years. It was certainly, at the start, the largest working parallel processor array in the world and achieved the fastest real-time image processing at that time.

7. Limitations of Image Processors

Every dedicated image processing system has its limitations. Most embody as much parallel structure as is practicable but every design falls short in some way or another. Special purpose circuits providing a very restricted range of functions can only be of similarly restricted applicability, although some attempts have been made to build computers combining several special purpose circuits into one composite system. Their performance is not impressive since most of the units are idle for most of the time and the effective parallelism is low.

The latency effect in pipeline processors together with the difficulty experienced in programming them in many applications has resulted in such systems falling into disuse. Processor arrays are also not easy to program although this is a skill which can be learned; there are no insurmountable difficulties in writing parallel forms of most image processing operations.

A more serious problem is that processor arrays suffer from two related inefficiencies. The first is that, in general, moving images in and out of the array is a serial process and therefore slow. Secondly, moving data between extremes of the array (as, for example, is necessary when performing Fourier Transforms) involves stepping through chains of neighbouring PEs and is also very slow. Both these inefficiencies can be lessened by adding more connection paths and this has been done in later systems, such as the Connection Machine [13]. A further problem is cost. Processor arrays are much less efficient when the size of the image array is larger than that of the processor array. Unless the level of integration can be made very high, the cost of

constructing and assembling enough PEs to match images of television quality is too great for the majority of potential users.

Research into parallel processing architectures for image processing has slowed down noticeably in recent years. On the one hand, the high cost of building these machines has made it difficult to obtain funding from the organisations which used to support this research. Equally, the long lead time for the production of new systems, taken together with the limited and uncertain market for the systems once they are produced, has discouraged manufacturing industry from continuing to be involved.

However, possibly the major factor which slowed the pace of this field of research was the lessening of demand from the image processing community. The widespread availability of high-powered workstations and the ever increasing performance/cost ratio of PCs have meant that the priority for development of systems with higher speed has been displaced by a need for more effective algorithms in the majority of active areas in applied image processing. It is also the experience of many in the field that the image processing software packages which can be purchased tend to be disappointingly inflexible, especially when there is a need to incorporate new functions not contained in the original package. Consequently, development effort has been switched from hardware to software.

A cynical comment on the state-of-the-art in image processing (or, perhaps more accurately, image analysis) would be that the computers now commercially available enable us to run bad programs adequately quickly and the use of even the best parallel processing methods would do nothing more than allow us to get poor results even faster. The same cannot be said about image generation, a wide-ranging subject embracing important and socially useful applications in the medical field as well as commercially profitable activities in computer games. In this area, there is always a demand for higher performance.

8. Predictions for the Future

Although the study of computer vision seems to be very unstructured and not progressing as well as had been optimistically expected three decades ago, there is still enough optimism amongst researchers to merit laying plans for the future when, it is believed, successful algorithms will have been developed and, once again, the need will be for faster processors. Enough is now understood about computer architecture to make it certain that adequately fast processing will only be achieved by the use of parallelism. At the same time, every attempt will have to be made to employ the fastest possible components.

There are physical limitations to the extent to which integrated circuit devices can be made faster. Current research is exploring these limitations by investigating nanotechnology where circuit components are defined with a precision approaching one nanometre (10^{-9} metre). If devices can be made to work with such dimensions,

then it would be conceivable that a CLIP4 array of size 512 x 512, together with adequate amounts of memory local to each PE, could be formed on one integrated circuit slice. Furthermore, images could be input to the slice by projecting them onto photosensitive components located with each PE. The power of such a system would far exceed anything now in use and the cost, assuming the technology had been given time to 'mature', would be a mere fraction of that of today's supercomputers.

Undoubtedly, there will be many major technical problems to solve. At this scale, long connections between parts of the array are difficult to fabricate. In particular, the distribution of control instructions synchronously across the array will be hard to achieve. Potential failure of devices embedded in the array will have to be combated by the liberal use of redundancy.

There are some indications that it may be hard to define and control the characteristics of the millions of devices in these giant arrays. If this is true, then a new style of programming might be necessary in which variability is not only accepted but also exploited. A Monte Carlo program running on a conventional computer gains its power to solve problems by introducing random numbers into what would otherwise be a completely predictable performance; could it be that a similar broadening of capability might be obtained by randomising the values of some of the device parameters in the processor arrays?

There is a philosophical point to be made here. We have always looked to human vision as a sort of rôle model for computer vision system designers but this may have been unwise. Human vision is there to enable humans to survive in their environment, not to equip humans with a precise optical measuring system. In everyday life, a broad, comprehensive view of the world is all that is needed and the speed at which this must be obtained is only of the order of human reaction time, i.e., an analysis in a few tens of milliseconds.

On the other hand, computer vision has generally been used to make fast and accurate measurements in a very constrained environment. This may imply that at least two very different types of image processing computer will be need: one in which speed and/or accuracy are the dominating goals and the other in which speed need not be of the highest but robustness in an unconstrained environment will be of fundamental importance.

Nevertheless, the ideas behind parallel processing computing are justified by the physiological example from which they sprang and that they were found to be effective when applied to computer architecture. It is difficult to conclude that tomorrow's computers will revert to a serial architecture. Parallelism is definitely here to stay.

References

1. Barnes, G.H., Brown, R.M., Kato, M., Kuck, D.J., Slotnick, D.L., Stokes, R.A.: The ILLIAC IV computer. IEEE Trans., Vol. C-17, (1968) 746-757
2. Batcher, K.E.: Design of a massively parallel processor. IEEE Trans., Vol. C-29, (1980) 836-840
3. Duff, M.J.B.: CLIP4: A large scale integrated circuit array processor. Proc. 3rd Int. Joint Conf. on Pattern Recognition, Coronado, CA, USA, (1976) 728-733
4. Duff, M.J.B., Jones, B.M., Townsend, L.J.: Parallel processing pattern recognition system UCPR1. Nuclear Instruments and Methods, Vol. 52, (1967) 284-288
5. Duff, M.J.B., Watson, D.M.: Automatic design of pattern recognition networks. Proc. Electro-Optics '71 Int. Conf., Brighton, England, Industrial and Scientific Conference Management Inc., Chicago, Ill., (1971) 369-377
6. Duff, M.J.B., Watson, D.M., Deutsch, E.S.: A parallel computer for array processing. In: Rosenfeld, J.L. (ed.) :Information Processing 74, North-Holland Publishing Co., Amsterdam, (1974) 94-97
7. Flanders, P.M., Hunt, D.J., Reddaway, S.F., Parkinson, D.: Efficient high speed computing with the Distributed Array Processor. In: Kuck, D.J., Lawrie, D.H., Sameh, A.H. (eds.): High speed computer and algorithm organization. Academic Press, NY, (1977) 113-127
8. Flynn, M.J.: Some computer organisations and their effectiveness. IEEE Trans. on Comput.,Vol.C-21, (1972) 948-960
9. Fountain, T.J.: Processor Arrays. Academic Press, London, (1987) 189-193
10. Golay, M.J.E.:Apparatus for counting bi-nucleate lymphocytes in blood. US Patent 3,214,574 (1965)
11. Golay, M.J.E.: Hexagonal parallel pattern transformations. IEEE Trans.on Comput., Vol.C-18, (1969) 733-740
12. Herscher, H.B., Kelley, T.P.: Functional electronic model of the frog retina, IEEE Trans., Vol. MIL-7, (1963) 98-103
13. Hillis, W.D.: The Connection Machine. MIT Press, Cambridge, Mass. (1985)
14. Hubel, D.H., Wiesel, T.N.: Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. J. Physiol., Vol. 160, (1962) 106-154
15. Lettvin, J.Y., Maturana, H.R., McCulloch, W.S., Pitts, W.H.: What the frog's eye tells the frog's brain. Procf. IRE, Vol. 47, (1959) 1940-1951
16. Levialdi, S.: CLOPAN: a closed-pattern analyser, Proc. IEE, Vol. 115, (1968) 879-880
17. McCormick, B.H.: The Illinois pattern recognition computer - ILLIAC III. IEEE Trans Elc. Comp., Vol. EC-12 (1963) 791-813
18. Preston, K. Jr.: The CELLSCAN system - a leukocyte pattern analyzer. Proc. Western Joint Computer Conf., (1961) 173-178
19. Slotnick, D.L., Borck, W.C., McReynolds, R.C: The SOLOMON.computer. Proc. AFIPS Fall Computer Conf., (1962) 87-107
20. Rosenblatt, F: Principles of Neurodynamics. Spartan Books, Wasington DC (1962)
21. Slotnick, D.L.: The fastest computer. Scientific American, Vol. 224(2), (1971) 76-87
22. Sternberg, S.R.:Cytocomputer real-time pattern recognition. Proc. 8th Automatic Imagery Pattern Recognition Symp., (1978), 205-214
23. Tanimoto, S.L.: Architectural issues for intermediate-level vision. In: Duff, M.J.B. (ed.): Intermediate-level image processing. Academic Press, London, (1986) 3-17
24. Unger, S.H.: A computer oriented toward spatial problems. Proc. IRE, Vol. 46, (1958) 1744-1750
25. Unger, S.H.: Pattern detection and recognition. Proc. IRE, Vol. 47, (1959) 1737-1752

Scheduling of a Hierarchical Radiosity Algorithm on a Distributed-Memory Multiprocessor

M. Amor, E. J. Padrón, J. Touriño, and R. Doallo

Dep. of Electronics and Systems, University of A Coruña
Campus de Elviña, s/n. 15071 A Coruña, Spain
E-mail: {margamor,emilioj,juan,doallo}@des.fi.udc.es

Abstract. This work presents an efficient implementation of a hierarchical radiosity algorithm on a distributed-memory multiprocessor. The parallel algorithm is based on a coarse–grain approach that avoids load imbalance by means of a dynamic scheduling strategy. Experimental results on the Fujitsu AP3000 multiprocessor using MPI show that this kind of architectures are appropriate to implement hierarchical radiosity methods as a stage of a image synthesis environment.

1 Introduction

Digital image synthesis is a field of computer graphics whose aim is the generation of realistic 2D digital images that emulate 3D objects. In order to achieve the desirable degree of realism, it is important to use global illumination algorithms that take into account the influence of each object located at the environment.

The radiosity method is a global illumination model widely used. The main advantage of this method lies in the fact that the obtained illumination results are independent of the viewpoint. Nevertheless, its drawback is the high computational cost, both in CPU time and memory requirements. For this reason, several approaches of the method have been proposed: progressive radiosity [2], hierarchical radiosity [6] and, more recently, wavelet radiosity algorithms [9]. This work is focussed on the parallelisation and scheduling of the hierarchical method. Although this method drastically reduces the complexity of the classical radiosity algorithm, it still has a significant computational cost, which justifies the use of parallel computing techniques.

In the literature, good results have been reported on shared-memory multiprocessors [10], where all processors have access to the whole scene, and the only bottleneck is the necessary control of R/W operations to avoid critical section problems and deadlocks. However, the results on distributed-memory multiprocessors are not so encouraging, mainly due to the communications overhead. Zareski *et al.* [11] applied fine–grain parallelism using a master-slave paradigm, where each slave performed the ray-polygon intersection computations on the corresponding subset of patches of the scene. In this case, the speedup of the algorithm is restrained by the bottleneck of having a master processor and the large number of communications required. Other implementations also follow a

master-slave model, but using a coarse–grain parallelism. In that case, each slave performs the whole computation of the radiosity on a group of patches of the scene, and the master takes charge of the dynamic patch distribution, as well as the convergence analysis. Among these implementations, one approach is to store the complete scene in the local memory of each processor [1], [3], in order to minimize communications, although large scenes cannot be processed due to memory requirements. Another approach is to distribute the scene among the processors [4], which allows to work with large scenes, although communications increase to a great extent.

In this work, we propose a parallel implementation of a hierarchical radiosity method on a distributed-memory multiprocessor. The scene is replicated in all the processors, and the load is dynamically balanced to avoid idle processors. We have used an SPMD (Simple Program Multiple Data) paradigm, that is, we do not waste one processor (master or scheduler) on load distribution tasks. Our scheduling is, therefore, distributed.

This work is organized as follows: next section describes the sequential algorithm of the hierarchical radiosity method; the parallel versions, both for a static load distribution and for a dynamic scheduling are presented in Section 3. Experimental results on the Fujitsu AP3000 multiprocessor are shown in Section 4. Finally, conclusions and future work are discussed in Section 5.

2   The Hierarchical Radiosity Algorithm

The radiosity method is based on applying to image synthesis the concepts of thermodynamics that rule the balance of energy in a closed environment. In fact, the radiosity method solves a global illumination problem expressed by Kajiya's equation [8], simplified by considering only ideal diffuse surfaces. The resultant equation system is:

$$B_i = E_i + \rho_i \sum_{j=1}^{n} B_j F_{ij} , \qquad (1)$$

where B_i is the radiosity of patch i, E_i is the emittance and ρ_i the diffuse reflectance. The summation represents the contributions of the other patches of the scene, where F_{ij} is the form factor between patches i and j. This factor represents the fraction of energy that leaves from a polygon and reachs directly another one. It is an adimensional constant that only depends on the geometry of the scene. The number of form factors between all pairs of n patches is $O(n^2)$, which makes traditional radiosity methods very expensive.

The complexity of the radiosity computation is dramatically reduced by using the hierarchical method. It subdivides the scene adaptively, applying the fact that small details are not significant at long distances; besides, the hierarchical method avoids computing some interactions if their form factors are zero, because it means that the patches cannot see each other. The scene is divided into patches (much larger than the ones used in the classical radiosity methods) that make
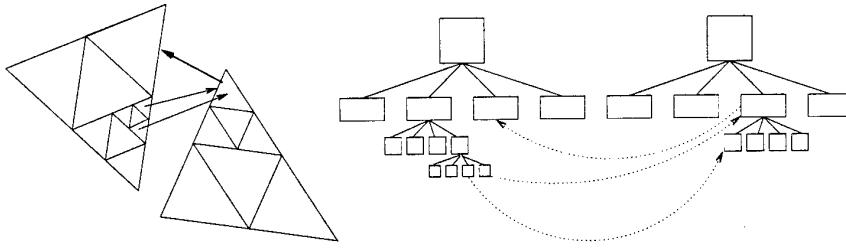
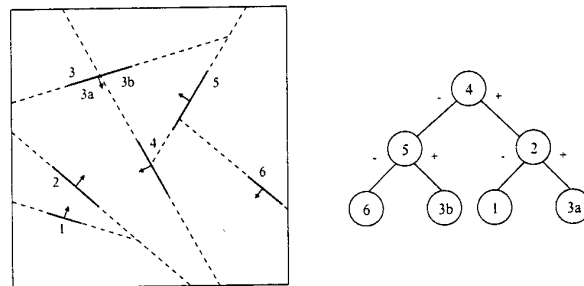**Fig. 1.** Interactions in an element hierarchy

Fig. 2. Example of a BSP tree (insertion order: 4, 2, 5, 1, 6, 3)

up the coarsest level of the hierarchy. These patches are successively subdivided into elements through an iterative process, until the desirable precision in the illumination of the scene is achieved (see Fig. 1).

The sequential algorithm of the hierarchical method based on [6] can be described as follows:

1. A BSP (Binary Space Partition) tree is built with the input polygons or patches (Fig. 2 shows an example). This tree will be useful later to determine the visibility between two patches in an efficient way.
2. For each patch inserted in the BSP tree, a list of initial interactions (or links) is computed. Each entry of this list has as destination other patch of the scene, potentially visible from the current patch. At this stage we consider that two patches are potentially visible if their positive sides are face to face. The form factor between the two patches involved is computed for each interaction. Once the initial interactions for all the patches have been computed, the iterative process that gathers the radiosity of each patch begins in the next step.
3. For each patch, the radiosity obtained from all its visible interactions is calculated. If the radiosity emitted by a certain link exceeds a given threshold,

the interaction must be refined (in this work, we use a BF refinement [5]). To perform this task, either the source element or the destination element of the interaction (depending on which of them has the largest area) is subdivided into a quadtree, where the children inherit the current radiosity of the father. The refinement is as follows:

(a) If the element to be subdivided is the source element of the interaction, four new interactions with each one of the children of the subdivided element are established in the destination element.

(b) If the element to be subdivided is the destination element, each one of its children inherits the interaction with the source element.

In both cases, the old interaction is discarded. For each patch, the radiosity of its current hierarchy of elements is computed through a post-order traversing of the quadtree.

4. Once all interactions between the elements of the scene have been processed, the complete radiosity of the scene is summed up and the convergence is checked by comparing this value with the result obtained in the previous iteration. If the convergence criterion is fulfilled, the algorithm finishes; otherwise, a new iteration begins in step 3.

3   The Parallel Algorithm

Two parallel versions of the hierarchical algorithm have been implemented, using the message-passing library MPI. Both versions are based on a coarse-grain approach, that is, each processor performs the whole computation of the radiosity for a set of patches of the scene. In the first approach, a static assignment of the patches to the processors, without applying any kind of scheduling, is carried out. Using this implementation, good results could be obtained for images that give rise to a regular load distribution among the processors. Nevertheless, in most cases the execution of the algorithm causes load imbalance due to the unpredictable behaviour of the refinement, which results in poor speedups; this fact is more significant as the number of processors increases. Thus, we have developed a second version of the parallel algorithm to balance the computations through a distributed dynamic scheduling. The following subsections describe both approaches.

3.1   Static Load Distribution

The first parallel algorithm we have developed distributes the workload among the processors so that, assuming n patches and p processors, each processor computes the radiosity for a fixed set of n/p patches. Next, the changes with respect to the sequential algorithm are detailed:

1. It is not worth parallelizing the BSP tree construction and the computation of the initial interactions because their execution times are negligible as compared with the whole radiosity algorithm. Thus, each processor generates its own local copy of the whole BSP tree. As new feature, once the patches are inserted in the BSP tree, they are sorted in decreasing area order.

2. Before beginning the iterative process, the sorted patches are cyclically assigned to the processors, in order to balance the load among the processors. That is, the patches whose order in the list is t, such that $t \bmod i = 0$ are assigned to processor i.

3. During the iterative process in which radiosity is computed, each processor only takes charge of its assigned patches. Besides, in each iteration, the processor keeps a record of those destination elements that correspond to patches assigned to a different processor.

4. Once the local calculation of radiosity in one iteration is completed, the processors start a global communication phase in which radiosity values and tree structures are updated. In this phase, each processor sends and receives data from the other processors, following an all-to-all communication pattern implemented by MPI total exchange routines (`MPI_Alltoall` and `MPI_Alltoallv`).

5. After the communication stage of each iteration, convergence is checked in parallel by means of a reduction operation (`MPI_Allreduce`). Each processor contributes the partial radiosity of its set of assigned patches to the reduction and, this way, the whole radiosity of the scene is obtained. Next, each processor compares this value with the radiosity in the previous iteration. As in the sequential code, if the difference is less than a fixed threshold, the iterative algorithm ends.

3.2 Distributed Dynamic Scheduling

The irregular and unpredictable behaviour in the execution of the hierarchical method makes the parallelisation using a static load distribution inappropriate, due to the appearance of load imbalance. Although we tried to overcome this problem by assigning cyclically a list of patches in order of area, it is not enough. A further approach could be a patch reassignment at the end of the first iteration based on the number of interactions of each patch. But this approach would not be very useful because we have experimentally checked that the first iteration is the most time-consuming and, therefore, it is necessary to solve the load imbalance from the beginning of the algorithm (specifically during the first iterarion). With this aim in view, we have developed a second parallel algorithm that implements a dynamic load distribution. The parallel algorithm (see Fig. 3) is summarized as follows:

1. Each processor builds its own BSP tree with all the patches of the scene, and sorts them in decreasing order of area.

2. The patches are cyclically distributed.

3. Each processor computes the radiosity of the assigned patches.

4. In the first iteration, if a processor finishes its corresponding computations, the next step is to check the presence of non-processed patches in the ordered global list. If so, the processor takes a set of k patches from the list, being k a parameter that is predefined experimentally depending on the problem size and the communication cost. We must take into account that high values
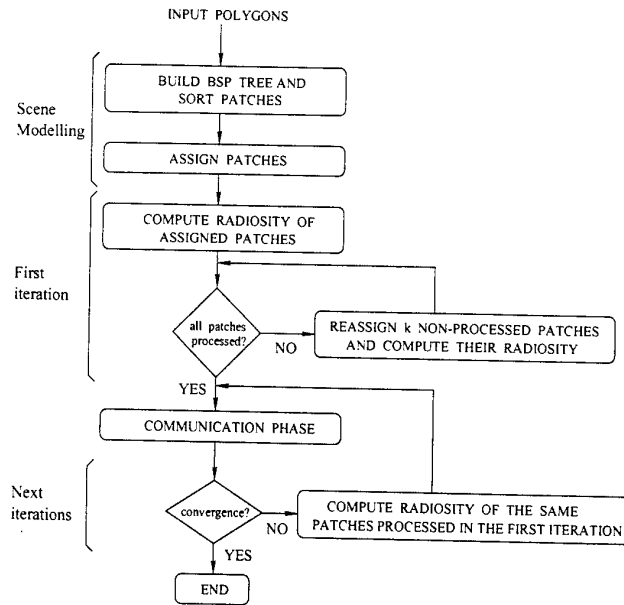
Fig. 3. Diagram of the algorithm using a distributed dynamic scheduling

of k reduce the communications overhead of the scheduling at the expense of a worse load balance, and vice versa. Step 4 is repeated until the list of non-processed patches is empty.

5. Once the radiosity of all the patches of the scene is computed, the communication phase takes place.
6. The convergence of the algorithm is tested in parallel and, in case of success, the algorithm finishes.
7. For the next iterations, each processor uses the same patches as in the first iteration (both the patches assigned statically and the ones taken from the list). At the end of each iteration, it returns to step 5.

As can be observed, the only significant difference between the static and dynamic implementations lies in the first iteration of the algorithm, specifically in the fourth step of the dynamic version. In this step, seemingly simple, the scheduling is carried out. The main drawback of this scheduling lies in the fact that two or more processors could compete for the same patch. Next, we describe in detail the protocol we have developed to overcome this problem.

Scheduling Protocol. In order to carry out a dynamic patch allocation, each processor must keep updated information about those patches that have not been still processed. This information is stored in the ordered list of patches and
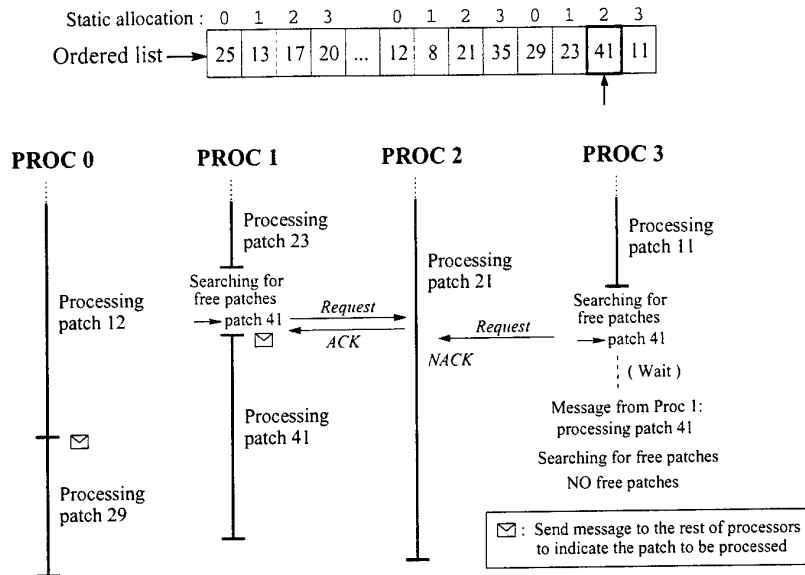
Fig. 4. Practical example of the protocol (assuming that $k=1$): processors 1 and 3 compete for patch 41, but only processor 1 gets the patch

must be available in all processors. As we are working in a distributed-memory environment, this availability is achieved by means of message-passing. Thus, before processing a set of patches, each processor communicates this state to the rest of processors. A drawback arises when two or more processors compete for the same patch. To avoid the assignment of the same patch to different processors, we have implemented a protocol based on making requests about the state of the patch that causes the conflict.

In the fourth step of the parallel algorithm described in this subsection, a processor, before taking a "free" patch (a patch that has not been still processed), sends a request message to the owner of that patch, that is, the processor that has the patch by means of the static assignment of step 2 (which is known by all the processors). If the owner of the patch is not still processing it, the ownership of the patch is transferred to the requesting processor (ACK) provided that the patch had not been still given to other processor. Otherwise, the patch request is refused (NACK). Note that, in this case, explicit messages are not used because the processor that is taking charge of the patch communicates this situation to the rest of processors.

Using this protocol, any kind of incoherence arising from the multiple assignment of one patch to two or more processors is avoided. For example, in Fig. 4 it can be observed that, once processors 1 and 3 have finished the computations associated with their assigned patches, they search for free patches in the or-
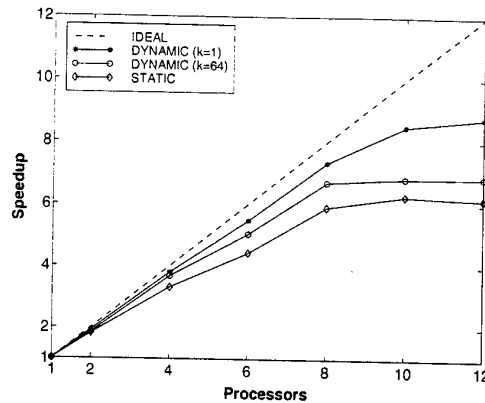
Fig. 5. Speedups for the test scene (650 polygons)

dered list of patches, beginning from the last patch of that list. Both processors try to get patch 41 (initially assigned to processor 2), but only processor 1 will finally get it; processor 3 carries on with the search of free patches in the list.

During this scheduling stage, nonblocking communications (both send and receive primitives) are used to overlap communication and computation. Besides, as the messages to be sent in this stage have the same format and size, as well as the same destinations, we have used persistent communications by means of MPI routines: `MPI_Send_init`, `MPI_Recv_init`, `MPI_Start` and `MPI_Request_free`. Therefore, the tasks involved in setting up the communication are accomplished only once.

4 Experimental Results

We have tested the parallel algorithms on the Fujitsu AP3000 multicomputer [7], whose nodes (UltraSparc-II processors at 300 Mhz) are connected via a high-speed communicacion network (AP-Net) in a 2D torus topology. The test scene is composed of 650 input triangles and is depicted in Figs. 6 and 7.

The results in terms of speedups are shown in Fig. 5, both for a static patch assignment and for the dynamic scheduling approach (using $k=1$ and $k=64$, and up to 12 processors). The execution time of the sequential algorithm is 374 seconds, and it is 42.68 using the dynamic scheduling on 8 processors. As can be observed, the speedup for the static case tends to be constant from 8 processors upwards due to the effect of load imbalance. The speedups are greatly improved using the dynamic scheduling that balances the load. This improvement is not so good for high values of k in relation to the whole scene size (for instance, $k=64$ in our example) because, although the number of communications decreases, the load imbalance becomes more noticeable and, therefore, the speedup results

come close to the static approach. For k <64, we have experimentally checked that the results are very similar to the ones obtained for k=1.

As can be observed, although better speedups are achieved by using the dynamic scheduling (for k=1), from a certain number of processors (10 in our example scene) the speedup does not rise accordingly. This is because the local computations assigned to each processor are not significant and it is not worth balancing small loads due to the communications overhead. Better speedups are expected for larger scenes because they involve more computations and, thus, the associated execution times are very high in relation to the communication factor. According to the speedup results we can conclude that the algorithm presents a reasonable scalability and the larger the scenes are, the more appropriate the dynamic scheduling is.

Regarding the correctness of the algorithm results (see the illuminated scene in Fig. 6 and the scene divided into elements in Fig. 7), we have used the residual error of the radiosity as error metric. We have experimentally confirmed that the error measured in the parallel implementations does not vary in comparison with the sequential version.

5   Conclusions

In this paper we have described a parallel implementation of the hierarchical radiosity method on distributed-memory architectures. The parallel method generates an irregular load distribution, which can be balanced following two strategies: on the one hand, the patches are initially distributed by area, trying to assign the same number of computations to each processor; on the other hand, a distributed scheduling performs a finer tuning to balance the load dynamically, by reassigning the smallest non-processed patches to the processors that finish their work. Good speedups, load balance and an acceptable scalability have been achieved through this approach.

We conclude that distributed-memory architectures can be efficiently used to implement the hierarchical radiosity method, although the main drawback is the memory overhead derived from the replication of the BSP tree, as well as part of the hierarchical structures of the elements, in each processor.

As future work, we intend to study alternative representations of the input 3D scene to avoid data redundance. We also expect to improve the iterative process for the radiosity computation, both to reduce execution times and memory requirements. Specifically, we will focus on the process to determine visibility, which is currently implemented by means of the ray-casting algorithm [5]. Our goal is to decrease the amount of time spent testing rays against the environment.
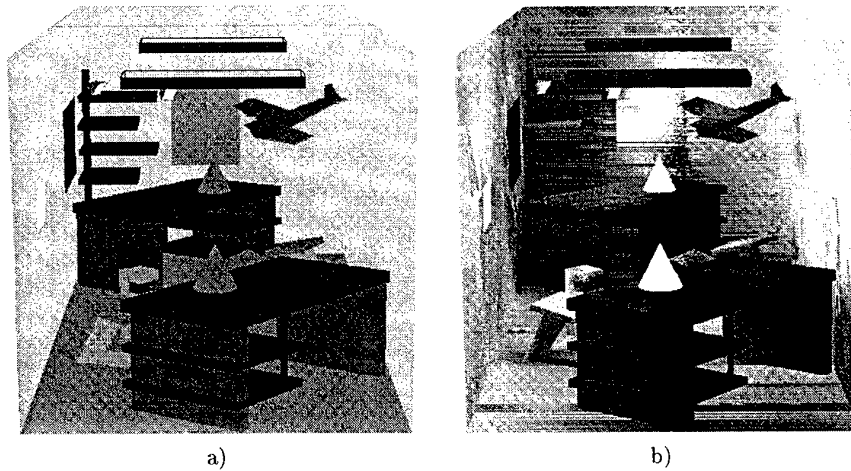
a) b)

Fig. 6. a) Scene before applying the hierarchical radiosity algorithm. b) Illuminated scene after applying the parallel algorithm
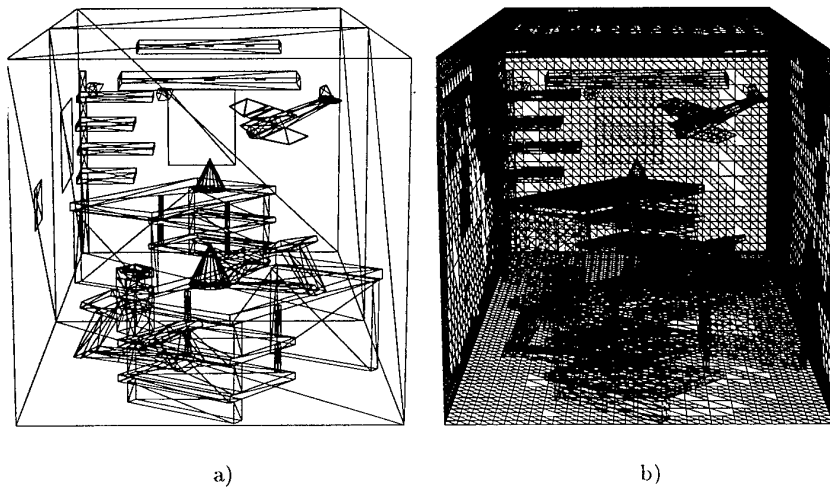

a) b)

Fig. 7. a) Patch division of the input scene. b) Final division of the scene into elements

References

1. Benavides, J.I., Cerruela, G., Trabado, P.P., Zapata, E.L.: Fast Scalable Solution for the Parallel Hierarchical Radiosity Problem in Distributed Memory Architectures. Second Eurographics Workshop on Parallel Graphics and Visualization, Rennes, France (1998) 49–57
2. Cohen, M., Chen, S.E., Wallace, J.R., Greenberg, D.P.: A Progressive Refinement Approach to Fast Radiosity Image Generation. Computer Graphics (SIGGRAPH'88 Proceedings) **22**(4) (1988) 31–40
3. Feng, C.-C., Yang, S.-N.: A Parallel Hierarchical Radiosity Algorithm for Complex Scenes. IEEE Parallel Rendering Symposium (1997) 71–78
4. Funkhouser, T.A.: Coarse-Grained Parallelism for Hierarchical Radiosity Using Group Iterative Methods. Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH'96 Proceedings) (1996) 343–352
5. Glassner, A.S.: Principles of Digital Image Synthesis. Morgan Kaufmann publishers (1995)
6. Hanrahan, P., Salzman, D., Aupperle, L.: A Rapid Hierarchical Radiosity Algorithm. Computer Graphics **25**(4) (1991) 197–206
7. Ishihata, H., Takahashi, M., Sato, H.: Hardware of AP3000 Scalar Parallel Server. Fujitsu Sci. Tech. J. **33**(1) (1997) 24–30
8. Kajiya J.T.: The Rendering Equation. Computer Graphics (SIGGRAPH'86 Proceedings) **20**(4) (1986) 143–150
9. Schröder, P.: Wavelet Algorithms for Illumination Computations. PhD Thesis, University of Princeton (1994)
10. Singh, J.P., Gupta, A., Levoy, M.: Parallel Visualization Algorithms : Performance and Architectural Implications. IEEE Computer **27**(7) (1994) 45–55
11. Zareski, D., Wade, B., Hubbard, P., Shirley, P.: Efficient Parallel Global Illumination Using Density Estimation. IEEE Parallel Rendering Symposium, Atlanta (Georgia) (1995) 47–54

Efficient low and intermediate level vision algorithms for the LAPMAM image processing parallel architecture

Domingo Torres[1], Hervé Mathias[2], Hassan Rabah[2], and Serge Weber[2]

[1]Programa de Graduados e Investigación en Ingeniería Eléctrica
Instituto Tecnológico de Morelia, México
Av. Tecnológico 1500 Morelia Mich., MEXICO
C.P.58600, Tel. 52 4 3121570 ext. 272, Fax. 52 4 3121643

[2]Laboratoire d'Instrumentation Electronique
Nancy (L.I.E.N), University of Nancy I France,
BP 239, 54506, Vandoeuvre Cedex, FRANCE
Tél : (33) 3 83 91 20 71, Fax : (33) 3 83 91 23 91

Abstract. This paper describes efficient parallel algorithms for low and intermediate level vision for a Linear Array of Processors with Multi-mode Access Memory (LAPMAM). Its special memory and its singular SIMD/restricted MIMD mode, combined with the parallel search and multiple update operation of the memory modules make LAPMAM very efficient in real time image processing. We have developed fast parallel algorithms to determine labeling, area and perimeter determination, histogram and median filter, we have taken advantage of LAPMAM characteristics for developer this efficient algorithms. The architecture and the algorithms were tested in language C and in a hardware simulator. The performance obtained are compared with that of different architectures.

1 Introduction

The computational demands of computer vision, which requires to process an enormous amount of information have incited a large number of research work and led to numerous architectures and algorithms [1] [2] [3]. The Sequential machines require an excessive amount of time. Hence this problem generally lies well beyond the capacity of existing sequential processors. Consequently, the possibility of the parallelism has been highly exploited.

In view of the number of processors and their topologies, parallel architecture may be classified into three-dimensional arrays (Pyramid, Hypercube, etc.), two-dimensional arrays (CLIP, Mesh with Reconfigurable Mesh, etc.), and one-dimensional array processors. The Electronic Instrumentation Laboratory of Nancy France is developing a linear array processor architecture for low an intermediate level vision that enhance its parallelism using a Multi-mode Access Memory (MAM) and a tree interconnection network. Also, its SIMD/restricted MIMD operating mode allows to LAPMAM

Candidate to the best student paper award

to switch between SIMD and MIMD mode automatically with a simple initial programming. In this architecture, a new concept of SIMD/restricted MIMD processors is also proposed. The processor has the SIMD structure with its typical advantages (simple implementation, high performance and no memory access conflicts), but also can work like a MIMD processor, taking a limited conditional decision with a simple control logic.

This article shows some fast parallel algorithms developed to take advantage of the LAPMAM characteristics [4]. We are obtained a quasi-optimal processor x time complexity [5] to the intermediate level vision algorithms. Concerning to the median filtering low level algorithm, the typical complexity of $O(n)$ to the 1-d architectures is obtained, but we show how the LAPMAM enhances the parallelism using the interprocessor communication to reduce the pixel computing operations. In the following section we present the organization of the LAPMAM. Then, we describe the algorithms developed. Finally we follow up with a discussion of the simulation results and a comparison of different architectures before concluding.
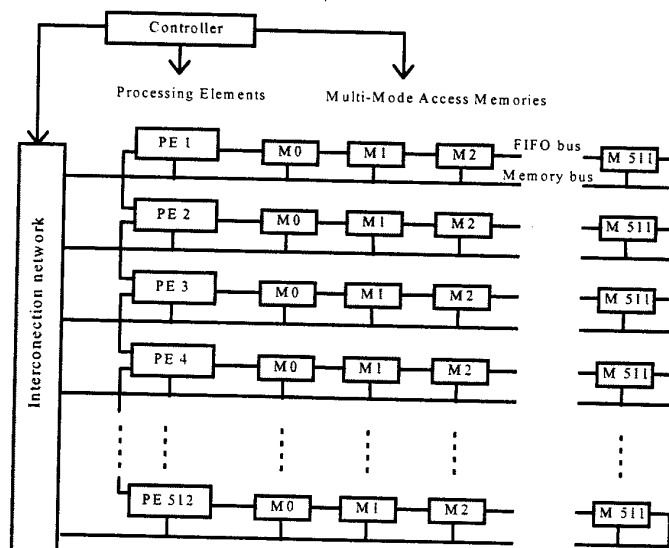

Figure 1: The LAPMAM architecture

2 LAPMAM architecture

The LAPMAM is a linear array of RISC SIMD/restricted MIMD processors with a Multi-mode Acces Memory. The LAPMAM has four memory planes that are dependent tanks to the bi-directional heteroassositive CAM property. A controller gives

the instructions word to each PE. The PE-PE and PE-MAM communications are carry out by a tree interconnection network and by a local bus (Memory bus) between the PEs and its corresponding row of memory modules.

The LAPMAM architecture for a 512 x 512 image (n=512) is shown on Figure 1. It features n processors organized in a linear array. Each of them is connected to a row of n memory modules. A special interconnection network allows every processor to reach any of the other processors and their associated memory rows. This network presents a tree structure and ensures global communication in $O(\log n)$ units of propagation time.

LAPMAM has four identical memory planes of log 512 bits denoted $M_{A1}[i,j]$, $M_{A2}[i,j]$, $M_{B1}[i,j]$ and $M_{B2}[i,j]$ ($0 \leq i,j \leq 511$). Each plane consists of 512 rows, each containing 512 memory modules. The four planes can be turned into two planes $M_A[i,j]$ and $M_B[i,j]$ of 2 log 512 bits. On Figure 1 the planes M_{A1}, M_{A2}, M_{B1}, M_{B2} are represented by the memory modules (M).

2.1 The Multi-mode Access Memory (MAM)

Our MAM module is basically a modified CAM. The CAM is a memory with addressing based on its *content*. This is an excellent solution in some applications where the RAM, with addressing based on its location, shows limited performance. The main advantage of the CAM is its capability to write/read a data to/from multiple locations in only one clock cycle or $O(1)$ time. Despite its relatively high cost, CAM has found since then enormous importance in various applications like data base management and image processing [6].

The CAM enhances the parallelism of an architecture because this memory works inherently in parallel. However, its utilization reduces the processing flexibility since the CAM can not be addressed by its position and the CAM reading is difficult. We have designed a CAM based memory with the possibility RAM and FIFO to solve the limitation of the CAM pure, it was called Multi-Mode Access Memory. The MAM modules constitute either four log n bits wide or two 2 log n bits wide memory planes. The four planes enable the architecture to work with algorithms that need to store intermediate results. The image loading procedure is also made simpler thanks to this possibility: an image frame may be stored in one memory plane while the previous is still under processing. The size of the memory words depends on the algorithms being run (2 log n bits for labeling and log n bits for median filtering for example). The CAM and RAM operation can be carried out in a whole plane, in a row (PE-MAMs) or in several rows of a plane. The FIFO operation is only carried out in the couples PE-MAMs.

Writing in normal CAM mode consists of simultaneously updating all the memory plane elements (M) with a New_Data where its content is equal to a Target_Data. The following algorithm describes the normal CAM mode:

forall M[address] ($0 \leq$ address $\leq$ n-1) **do_in_parallel**
   if (M[address] = = Target_Data)
      M[address] = New_Data;

endif
endforall

Writing in interactive CAM mode consists of updating elements of a memory plane with a New_Data if the content of the elements of a different plane is equal to a Target_Data. The interactive CAM mode is of highest interest as, in this mode, two planes can be dependent on one another. On Figure 2 a PE addresses plane A with a target_Data=1 (corresponding to "objet 1") to update the corresponding memories in plane B with a "blue" data in O(1) time.
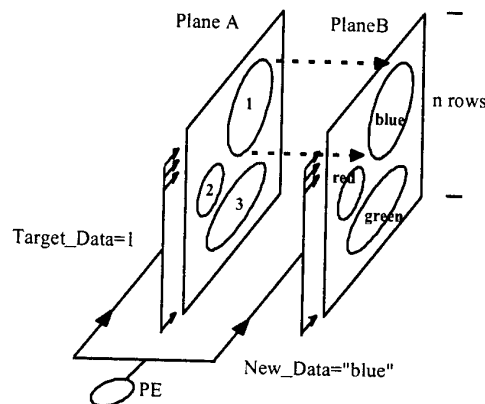

Figure 2: The capability of a PE for writing to multiple rows in the interactive CAM mode, in *O*(1) time.

The FIFO mode is used to perform a circular left/right data movement in a MAM-PE row. Two clock cycles are required to transfer the four planes. This mode allows the PE to get a neighboring information between the memory modules which is absent in ordinary CAM cells. Furthermore, the PE being part of the FIFO, it can read a new memory data at the same time that it writes in its memory row the data processed. The FIFO mode thus allows dividing the number of data access by two.

The RAM mode is obtained using the interactive CAM mode. A different address must be stored in one plane of each memory module. A subsequent interactive CAM operation with the desired location will only activate one memory module. The MAM planes not used to store the address may then be either read or written using the interactive CAM mode.

2.2   The processing element (PE)

The processing element is a RISC SIMD processor with the possibility of take some decisions. Each PE can be activated or deactivated independently. The PE is able to compute a basic logical or arithmetic operation in $O(1)$ time. It can communicate in $O(1)$ units of propagation time with its adjacent PEs or with its associated memory modules. Furthermore, it can communicate in $O(\log n)$ units of propagation time

either with its non-adjacent PEs or its non-adjacent memory modules through the interconnection network.

When the processor is connected directly to its memory row, the access to the data contained in this line is accomplished by means of the FIFO, RAM and CAM modes. In the FIFO mode the data of the PE are transferred to the last MAM module of the row.

Using the tree interconnection network, a processor can be connected to several memory rows or even to all memory rows. This depends on the interconnection network programming. To enable the communication between PEs, each PE has a data output toward its adjacent PEs (upper PE, lower PE).

2.3   Restricted MIMD mode

A SIMD PE is characterized by its reduced size. But, because it does not have a unit control, these types of PEs can not take internal decisions. Then, to execute different operations on different data, an architecture SIMD has to connect and disconnect the processor as many times as the number of different operations. On the other hand, the MIMD processor, that has a unit control, can take internal decision, but they are very much complex. It limits the number of PEs in an integrated circuit. The LAPMAM architecture has a SIMD processor that can take some internal decisions, this possibility increment the flexibility of this architecture avoiding the connection and disconnection of PEs pour perform different instructions, reducing the computing time. This particular characteristic is called by us restricted MIMD mode because the processor can only take a few decisions.

2.4   The interconnection network

The LAPMAM interconnection network performs the communications PE-PE or PE-MAMs in $O(\log n)$, but in some case it can be executed in $O(1)$, possibility that we are exploited in our algorithms. Moreover, this network has the characteristics of modularity and extensibility that allow to the network to be constructed from a small set of basic modules and to be extended to a larger size. These possibilities are very interesting for a VLSI implementation.

The interconnection network is reconfigurable by $n+(3n/4)-1$ switch modules denoted S_g. Each switch S contains (4 log 512 + 2) three states buffers. The PE-PE and PE-MAMs connections can be carried out in regions. Some PEs can be connected to a region of 2, 4, 8 etc. elements (PEs or rows of MAMs). This connection allows certain PEs to do a regional or global communication in $O(1)$ with a propagation delay of $O(\log n)$. But, in general, a global communication time of $O(\log n)$ is obtained with this type of network. A tree interconnection network for an architecture with eight PE is presented on Figure 3.
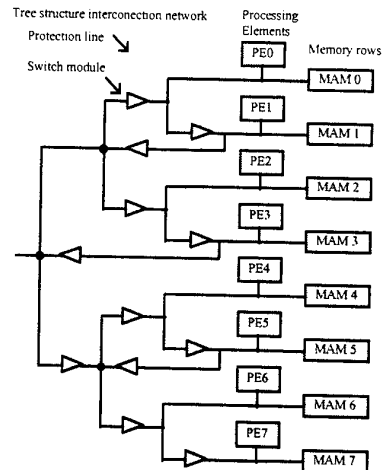
Figure 3: Tree structure interconnection network for LAPMAM architecture with 8 PE and 8x8 MAMs.

3 Algorithms

This section present the efficient parallel algorithms used to evaluate the LAPMAM architecture. The algorithms developed are connected components labeling, area and perimeter of a region, histogramming and median filtering. A description of these algorithms for an image n x n is done in the following paragraphs.

3.1 The connected components labeling

Labeling consists in assigning a unique label to the connected components in the image. It is a fundamental task in image processing and a lot of architectures and algorithms have been created to solve this problem [5]. Our algorithm, which is based on a divide and conquer technique, leads to a complexity of $O(n \log n)$. We remark that this complexity is independent of the shape of object and the type of image, it could be black and white or level of gray, a 4-connectivity image is assumed. We suppose that an initial image is available in $M_A[i,j]$ while the $M_B[i,j]$ plane is initialized at 0. This algorithm is comprised of two stages as follows.

Row processing: The values of $M_A[i,j]$ and $M_A[i+1,j]$ of a given row, starting with i=0, are tested. If both are identical, the $M_B[i+1,j]$ is assigned by the $M_B[i,j]$ value. Otherwise, the $M_B[i+1,j]$ is assigned by its row+1 value. This operation is done in parallel for each row and is repeated by first incrementing the index i. At the end of row processing (i=n), each *objet* is labeled according to its smallest value. The value in each row_{MBj} represents a label of pixel in row_{MAj}. The complexity for this stage is $O(n)$.
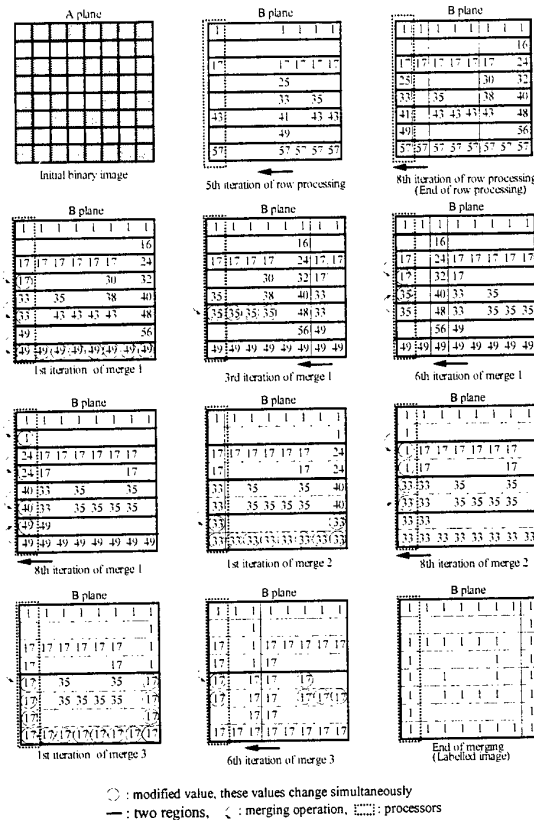
Figure 4: Labeling example

Merging: The values of $M_A[i,j]$ and $M_A[i,j+1]$ of two given rows, starting with i=0, are tested. If they are equal, the largest value addresses row_{MBj} and $row_{MB(j+1)}$ in normal CAM mode to update all CAMs in the 2 rows with the smallest value. This is called the *broadcast mode* and takes $O(1)$ time. Otherwise, there is no operation. The operation is carried out for each row in parallel and is repeated by incrementing i until the 2 rows are merged (i=n). The 2 merged rows are henceforth called *region*. The merging is repeated by activating the following stage of the tree structure to form 2 larger regions. Two adjacent boundaries of both regions are then scanned and compared. The same procedure as that of above processing is undertaken. The largest and smallest values are then transferred in the broadcast mode but this time in order to merge the 2 regions instead of the 2 rows. The merging is repeated until the last stage of the tree structure. The total number of stages reaches log n and each merge takes n iterations, it so follows that the total merging takes $O(n \log n)$ time. At the end of the procedure, the background is defined, with a simple CAM operation all the objects

with the value chosen become the background, they will be labeled 0. The **Figure 4** shows a labeling example, it considers only an object for clear demonstration. The initial image is shown in the first frame, the following two frames (left to right), present the row processing. In this part there is a circular data in each row that is executed by the FIFO instruction. Each data pass by the PE for be treated until all the data get their original organization. The Merging stage is executing using also the function FIFO to provide the data memory to the each PE. The data actualization is done by the CAM function as is shown in all the merging frames. The three stages (log 8) of merging are essentials for processing an 8x8 image. The final result (image labeled) is presented in the last frame.

The algorithmic description of our labeling method is presented in the following paragraphs. The used terminology is:

- $M[i,j]$: memory module in column i and row j.
- $M[*,j]$: memory module in all columns and row j.
- $M_{(CAM)}[i, j]$: CAM mode.
- In the interactive CAM mode, writing $M_A[*, j]$ with a New_Data if $M_B[*,j]$ is addressed by a Target_Data is shown as follows:

    forall CAMs $M_{B(CAM)}[*, j]$, ($0 \leq j \leq n-1$) **do_in_parallel**
    if ($M_{B(CAM)}[*, j] == Target\_Data_L$)
      $M_{A(CAM)}[*, j] = New\_Data_L$;
    endif
    enforall

Algorithm: Region Labeling

Input: Initial image in $M_A[i,j]$.
Output: Labeled image in $M_B[i,j]$.

Initialization of memory:
forall Memory $M_B[i,j]$, ($0 \leq i \leq n-1$, $0 \leq j \leq n-1$) **do_in_parallel**
    $M_B[i,j]=0$;

Row processing:
forall Processors P_j, ($0 \leq j \leq n-1$) **do_in_parallel**
    for (i=0 ; i $\leq$ n-1 ; i++)
    if ($M_A[i,j] == M_A[i-1,j]$) $M_B[i,j] = M_B[i-1,j]$;
    else
    $M_B[i,j] = i+nxj+1$;
      // i+nxj+1: row major initialization (1, 2, 3,.., n)
    endif
    endfor
endforall

Merging:
for (f=1; f $\leq$ log n ; f++)
    for (i =0 ; i $\leq$ n-1; i++)

```
      forall Processors P_i, (L=2^{r-1}(2k-1)-1, 1 ≤ k ≤ n/2^r) do_in_parallel
   if ( M_A[i,L] = = M_A[i,L+1] )
       if ( M_B[i,L] < M_B[i,L+1] )
             Target_Data_L = M_B[i,L+1];
             New_Data_L = M_B[i,L];
       else
             Target_Data_L = M_B[i,L];
             New_Data_L = M_B[i,L+1];
       endif
             forall CAMs M_{B(CAM)}[*, r+2^r(k-1)],
             ( 0 ≤ r ≤ 2^r-1, 1 ≤ k ≤ n/2^r ) do_in_parallel
                 if ( M_{B(CAM)}[*, r+2^r(k-1)] = = Target_Data_L )
                     M_{B(CAM)}[*, r+2^r(k-1)] = New_Data_L ;
             endif
           endforall
        endif
         enforall
   endfor
endfor
```

Background definition:
```
    forall CAMs M_{B(CAM)}[*, j], ( 0 ≤ j ≤ n-1) do_in_parallel
  if ( M_{B(CAM)}[*, j] = = 0 )
    M_{B(CAM)}[*, j] = 0 ;
  endif
    endforall
```

3.2   Area or perimeter determination

These two algorithms are very similar to the precedent one. They also use the divide-and-conquer technique and are carried out in the same two phases. The only difference is the type of processing which affects the pixels. For the area determination, all the pixels with a given label are counted while only those situated on the boundary of the connected components are taken into account in the case of perimeter determination. Here again, these algorithms have a complexity of $O(n \log n)$.

3.3   Histogramming

The histogram of an image is defined as the total number of pixels belonging to each gray-level value. For the histogram determination we use the organization technique of results proposed by Alnuweri [7]. In this algorithm, an initial image is supposed to be available in the A1 plane. The B1 plane that is initialized at 0 is used to store the result of histogramming in which its columns correspond to the gray-level values while its rows correspond to the number of pixels.

Row processing: Here, at each iteration i, each PE reads a gray-level value P in the ith column of the A1 plane. The value Q, in the Ath column of the B1 plane, is then incremented. Since each incrementing operation takes $O(1)$ time and there are n iterations, therefore this phase takes $O(n)$ time.

Sum-on-tree: Here, at each iteration i, each PE reads the value of the ith column in the B1 plane. The sum-on-tree operation is employed to add all values stored in the PEs. Since each sum-on-tree operation takes $O(\log n)$ time and there are $O(n)$ iterations, this phase takes $O(n \log n)$ time.

Hence, the complexity of our histogramming is $O(n \log n)$ which is optimal for $G \leq n$, where G is the number of the gray-level value.

3.4 Median filtering

Median filtering consists of replacing each pixel of a given image by the median of the pixels contained in a window centered around that pixel [8]. This filtering operation is useful in removing isolated lines or pixels while preserving spatial resolution. The classic method consists of sorting the elements from the smallest to the largest in a value table. The 5th element, in the case of 9 elements, will be the median value. To sort them, comparisons of two by two elements are done executing a permutation to change their place in the table. In this method, all the window pixels are accumulated in the PE registers. The PE uses others registers to execute the sorting operations and to store the results. We propose a fast filter median 3x3 algorithm that uses only some PE registers thanks to the interprocessor communication. This algorithm can be extended to larger windows. The pixels of the window are distributed as is shown on Figure 5.

The algorithm consists of three steps: first, sorting in parallel the 3 data groups stored in the PEs from the smallest to the largest. In the second step, the largest pixel between the smallest group of each row is found. In the same way, the smallest pixel among the largest ones of each row is found and the median of the medians group is detected too. The final stage consists of detecting the median value of the diagonal integrated by the elements found in the second part, it will be the final median value of this group of pixels. Figure 5 shows the pixel array and the three stage of the method.

The relevant characteristics of this method are that the classification of the three elements obtained by a PE, in the first step, is used by its two immediate PE neighbors. Then, the median filter operations of this part are divided by three. The second step consists only in a few operations because it is not necessary to sort the whole column. It is enough to find the smallest, largest and median value in the corresponding columns. It is the same case for the third step: the final median value is detected with 4 comparisons. In conclusion, the LAPCAM system realizes the 3x3 median filtering in $O(n)$ steps, as many other linear architectures, but our algorithm permits to process each pixel in only sixteen operations.
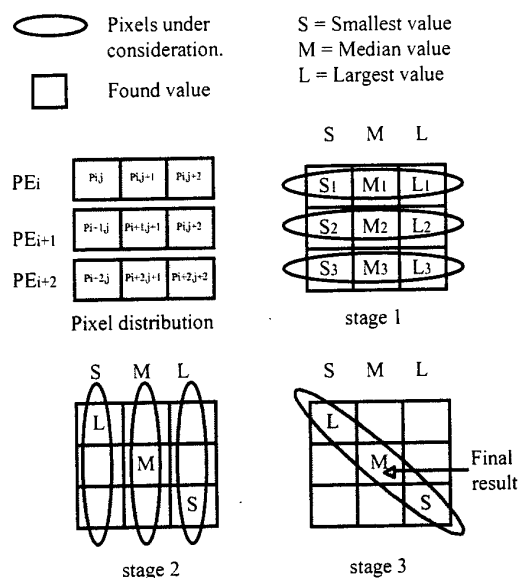
Figure 5: Pixel classification window used to find the 3x3 median value.

4 Performance

To validate these algorithms we have done a hardware simulation using VLSI tools of a LAPMAM prototype with 8 PE. All the algorithms mentioned above were implemented on this prototype. The architecture simulation was done at 50 MHz frequency. All the algorithms were simulated at this frequency, the results were extended to an 512 PE architecture, they are showed in Table 1. LAPMAM computes these low and intermediate-level image processing algorithms much faster than the video rate. The best performance results of the DARPA II image understanding benchmark [9] for the algorithms evaluated are compared in the first part of the Table 2. The architectures included are the Connection Machine (CM) with 64 K of PE, the Associative String Processor (ASP) that has 262,144 processors and the Image Understanding Architecture (IUA) that consists of three difference processors: low level SIMD PEs (processor-per-pixel), 4096 intermediate level SIMD/MIMD 16 bits processors, and one high level multiprocessor. For the tasks compared, our architecture is among the best ones while being the least complex. On this benchmark, only IUA has better results for labeling. However, it features for many more processors than our architecture. Otherwise, LAPMAM has the best computation times. This does not necessary mean that our architecture is much better than the others, since these architectures are very different and the technology evolution is not considered. Nevertheless, it gives a good idea of the LAPMAM's potential in low and intermediate level tasks.

Table 1: LAPMAM estimated performance for a 512x512 image

Agorithm	Complexity	Steps	time (μs)
Labeling	O(n log n)	30737	614.74
Area of a region	O(n log n)	43548	870.96
Perimeter of a region	O(n log n)	46091	921.82
Histogram	O(n log n)	21015	420.3
Median filter	O(n)	10241	204.82

In the second part of the Table 2, the LAPMAM estimated performance is compared with architectures that are more similar to LAPMAM: VIP [10], SliM-II [10] and IMAP VISION [11]. In this comparison, our architecture has the best results for these algorithms. Its enhanced parallelism allows the reduction of the algorithms complexities. The use of CAM and the tree structure of switches in interconnection network make the LAPMAM extremely efficient in terms of connected component analysis and median filtering tasks. However, because of the MAM modules, the architecture is more complex than the ones that use RAM. LAPMAM thus necessitates a full custom approach for its hardware implementation

Table 2: The LAPMAM estimated time results compared with others architectures (time in ms)

Algorithm	DARPA II Benchmark results for a 512x512 image			LAPMAM similar architectures			LAPMAM 50 MHz, 512 PEs, 512x512 image
				VIP	SliM-II	IMAP-VISION	
	CM 64 K	ASP	IUA	1024 PEs, 50 MHz	512 PEs, 40 MHz	512 PEs, 40 MHz, 256x240 image	
Labeling	100	22.8	0.0596	-	-	19.5*	0.614
Median filter	15	0.72	0.5625	3.672	2.525	1.07	0.204
Histogram	-	-	-	-	3.313	1.33	0.420

* Worst-case example

1   Conclusion

Fast parallel algorithms for labeling, area, perimeter, histogramming and 3x3 median filtering have been developed in a new parallel architecture dedicated to image processing. The quasi-optimal processor x time complexity of these algorithms and the efficient utilization of the MAM had demonstrated the interest of this architecture for low and intermediate level vision, particularly for connected component analysis and median filtering. The use of a tree structure of switches has proved to be an excellent solution to decrease the reduction of data propagation time in interconnection net-

Candidate to the best student paper award

work. Considering the algorithms results, the system presents very good performance for real time image processing. This will be confirmed with the development of other algorithms and the system hardware implementation. Another algorithms and a LAPMAM prototype VLSI are under development at the moment.

References

[1] V. K. P. Kumar, *Parallel Architectures and Algorithms for Image Understanding*: Academic Press INC., 1991.

[2] M. Maresca, "Special Issue on Parallel Architectures for Image Processing," *Proceeding of IEEE*, vol. 84, pp. 915-1049, 1996.

[3] A. Downton, and D. Crookes, "Parallel Architectures for Image Processing," *Electronics & Communication Engineering Journal*, vol. 10, pp. 139-151, 1998.

[4] D. Torres, Herve Mathias, Hassan Rabah, and Serge Weber, "SIMD/restricted MIMD Parallel Architecture for Image Processing Based on a New Design of a Multi-mode Access Memory," presented at International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA'99, Las Vegas Nevada USA, 1999.

[5] H. M. Alnuweiri, "Parallel architectures and algorithms for image component labeling," *IEEE Transactions On Pattern Analysis and Machine Intelligence*, vol. 14, pp. 1014-1034, 1992.

[6] L. Chisvin, and Duckworth R. J., "Content-Addressable and Associative Memory: Alternatives to the Ubiquitous RAM," *Computer*, vol. 1, pp. 51-64, 1989.

[7] H. M. Alnuweiri, and V.K. Prasanna Kumar, "Optimal image computations on reduced processing parallel architectures," in *Parallel Architectures and Algorithms for Image Understanding*, V. K. P. Kumar, Ed. New York: Academic Press, 1991, pp. 157-183.

[8] D. Helman, and J. Jájá, "Efficient Image Processing Algorithms on the Scan Line Array Processor," *Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, pp. 47-56, 1995.

[9] C. Weems, Riseman Edward, and Hanson Allen, *"The DARPA Image Understanding Benchmark for Parallel Computers,"* Journal of Parallel and Distributed Computing, vol. Vol. II, fasc. 1, p.p. 1-24, 1991.

[10] H. C. Chang, Soohwan Ong, and Myung H. Sunwoo, "A linear Array Parallel Image Processor: SliM-II," presented at Proceeding IEEE, International Conference on Aplications-Specific Systems, Architectures and Processors, Zurich, Switzerland, 1997.

[11] Y. Fujita, N. Yashamita, and S. Okazaki, "IMAP-VISION: An SIMD Processor with High-Speed On-Chip Memory and Large Capacity External Memory," presented at MVA'96 IAPR Workshop on Machine Vision Application, 1996.

Candidate to the best student paper award

Parallel Image Processing System on a Cluster of Personal Computers

J. Barbosa*, J. Tavares**, and A.J. Padilha

FEUP-INEB, Praça Coronel Pacheco, 1, 4050 Porto (P)
e-mail: jbarbosa@fe.up.pt

Abstract. The most demanding image processing applications require real time processing, often using special purpose hardware. The work herein presented refers to the application of cluster computing for off line image processing, where the end user benefits from the operation of otherwise idle processors in the local LAN. The virtual parallel computer is composed by off-the-shelf personal computers connected by a low cost network, such as a 10 Mbits/s Ethernet. The aim is to minimise the processing time of a high level image processing package. The system developed to manage the parallel execution is described and results obtained for the parallelisation of high level image processing algorithms are discussed, namely for active contour and modal analysis methods which require the computation of the eigenvectors of a symmetric matrix.

1 Introduction

Image processing applications are computationally demanding due to the amount of data to be processed, to the response time required, or due to the complexity of the image processing algorithms. A wide range of hardware has been used for image processing. For low level image analysis, where each processor performs a uniform set of operations based on the image data matrix in a fixed amount of time, SIMD computers using data parallelism may be used; in [28] a special purpose SIMD computer with 1024 processors was presented. Systolic Arrays [11] which can exploit the regular and constant-time operations of an algorithm are also a possible option.

For high level image processing, e.g. pattern recognition, where each processor is assigned an independent operation, MIMD supercomputers commonly used in simulation have been used [3]. For real time vision applications special MIMD computers were developed, e.g. ASSET-2 based on PowerPC processors for computation and on Transputers for communication [29]. MIMD supercomputers were characterised by allowing a diversity of structures, however, technological factors have been forcing a convergence towards systems formed by a collection of essentially complete computers connected by a communication network [9]. The processors of these computers become the same ones used in

workstations. Therefore, the idea of forming a parallel computer from a collection of off-the-shelf computers comes naturally, and fast communication techniques were also developed for that purpose [25]. Several cluster computing systems have been developed, e.g. the NOW project [2].

Our aim is not to build a cluster of personal computers for parallel processing but to perform parallel processing on already existing group clusters, where each node is a desktop computer running the Windows operating system. These clusters are characterised by having a low cost interconnection network, such as a 10 Mbits/s Ethernet, connecting different types of processors, of variable processing capacity and amount of memory, thus forming a heterogeneous parallel virtual computer. Due to network restrictions, which do not allow simultaneous communication among several nodes, the application domain is restricted to one or two dozens of processors.

The motivation for a parallel implementation of image algorithms comes from image and image sequence analysis needs posed by various application domains, which are becoming increasingly more demanding in terms of the detail and variety of the expected analytic results, requiring the use of more sophisticated image and object models (e.g., physically-based deformable models), and of more complex algorithms, while the timing constraints are kept very stringent.

A promising approach to deal with the above requirements consists in developing parallel software to be executed, in a distributed manner, by the machines available in an existing computer network, taking advantage of the well-known fact that many of the computers are often idle for long periods of time [20]. It is quite common in many organisations that a standard network connects several general purpose workstations and personal computers, accumulating a very substantial computing power that, through the use of appropriate managing software, could be put at the service of the more computationally demanding applications.

Existing software, such as the Windows Parallel Virtual Machine (WPVM) [1], allows building parallel virtual computers by integrating in a common processing environment a set of distinct machines (nodes) connected to the network. Although the parallel virtual computer nodes and the underlying communication network were not designed for optimised parallel operation, very significant performance gains can be attained if the parallel application software is conceived for that specific environment.

2   Image Algorithms and Systems

The image algorithms that have been parallelised consist of a set of low level image processing operations namely edge detection [27,6], distance transform, convolution mask, histogramming and thresholding, whose suitability to the cluster architecture was analysed in [4]. A set of linear algebra algorithms required for high level image processing was also implemented. The algorithms are the matrix product [14], LU factorisation [7], tridiagonal reduction [8], symmetric QR iteration [15], matrix inversion [23] and matrix correlation.

In this paper, results focus on high level image processing algorithms, namely active contours [19] and modal analysis [26].

Some image processing systems have been proposed to run on a cluster of personal computers. In [17] two highly demanding vision algorithms were tested giving superlinear speedup, due to memory pagination on one workstation. The machines formed an homogeneous computer. In [18] a high level interface parallel image processing library is presented and results for low level image operations on an Ethernet network of HP9000/715 workstations and an ATM network of SGI workstations are reported. In [21] a machine independent methodology was proposed for homogeneous computers; results were presented separately for two SMP workstations with two and eight processors, not requiring communication between machines.

Our implementation differs from the ones mentioned above due to the consideration of a general bus type heterogeneous cluster where data is distributed in order to obtain a correct load balancing and the number of processors that participate in a distributed algorithm vary dynamically in order to minimise the processing time of each operation [5].

3   The System Architecture

The computers that belong to the virtual machine run a process to monitor the percentage of processor time spent with the local user. Conceptually, local users have priority over the distributed application and the computer will not be available if the mean local user time is above a minimum threshold during a specified period of time, e.g. 5 seconds.

Each algorithm or task is decomposed until indivisible operations are obtained to which parallel code exists. When a parallel algorithm is launched the master process schedules work to the processors of the virtual machine according to their availability and choosing a number of processors that minimise the processing time of individual operations, allowing data redistribution if the optimal grid [4] of processors changes from operation to operation.

As an example, the algorithm to extract the contour of an object can be decomposed into edge enhancement, thresholding and contour tracking operations.

3.1   Hardware Organisation and Computational Model

The hardware organisation is shown in figure 1. Each node of the virtual machine is a personal computer under the Windows NT operating system, running WPVM software to communicate. The interconnection network is an Ethernet at 10/100 Mbits/s.

Several computational models [9, 30, 16] were proposed in order to estimate the processing time of a parallel program in a distributed memory machine. Although they could be adapted for the cluster of personal computers, a specific and simplified model is presented below.
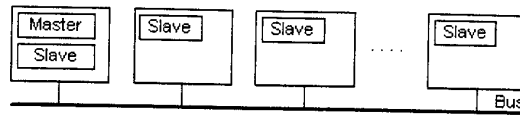
Fig. 1. Hardware organisation

Each node of the machine is characterised by the processor capacity S_i, measured in Mflop. The network is characterised by allowing only one message to be broadcast at a given time, the latency time (T_L) and the bandwidth (LB). The time to send a message (T_{Comm}) composed by nb bytes is given by:

$$T_{comm} = T_L K + \frac{nb}{LB}, \qquad K = \left\lceil \frac{nb}{packetsize} \right\rceil \tag{1}$$

The value K multiplies T_L due to the partition of each message into packets of length 46 to 1500 bytes (*packetsize*), existing a latency time for each packet; 1024 is a typical packet size.

The parallel component T_P of the computational model, equation 2, represents the operations that can be divided over a set of p processors obtaining a speedup of p, i.e. operations without any sequential part.

$$T_P(n,p) = \frac{\psi(n)}{\sum_{i=1}^{p} S_i} \tag{2}$$

The numerator $\psi(n)$ is the cost function of the algorithm measured in floating point operations (*flop*) as a function of the problem size n. For example, to multiply square matrices of size n, the cost is $\psi(n) = 2n^3$ [10].

3.2 Software Organisation

Each operation is represented by an object containing the parallel and serial implementation of the code, since the system can schedule a sequential execution remotely if it is advantageous. The object associated to the operation also contains the computational complexity and the amount of data required to exchange in order to complete the operation. Based on these parameters the system determines the number and which processors minimise the operation processing time [4].

Each data instance to be processed, an image or a matrix, is represented by an object responsible for accessing data items correctly according to the data distribution information.

Data distribution is represented by independent objects with functions to locate any item of data and to translate global to local indexes and vice-versa. Each object can be shared by more than one data instance. Figure 2 shows the software organisation.
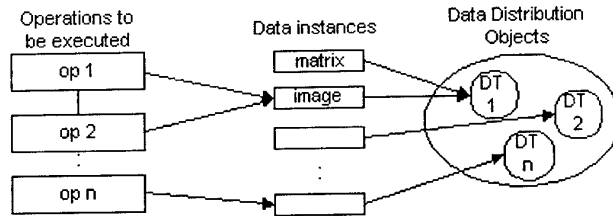
Fig. 2. Software organisation

The user describes a macro of sequential operations to be executed referring the data instances to be processed. The system executes each operation in parallel determining for each one the number of processors to be used in order to minimise the processing time. The data distribution suitable for each operation is codified in the operation code.

```
Input i1 image1.bmp
Shencastan i1 i2 i3 0
Histogram i2 outfile.txt i4
Output i2
Output i4
```

Fig. 3. Macro describing the operations to be executed

Figure 3 shows an example of a macro. To the input file *i1* an edge detector [27] is applied, the operator output, the magnitude and direction, being stored in *i2* and *i3* respectively. The histogram is then computed and displayed as an image, being also saved in a text file.

3.3 Data Distribution and Load Balancing

Different strategies are applied to images and matrices. Images are partitioned in blocks of contiguous rows or columns and the blocks are assigned to each process [4]. This distribution is suitable for data independent image operators. The matrices are organised in square blocks of data and a heterogeneous adapted version [5] of the block cyclic domain distribution [13] is used to assign them to the processor grid.

A balanced distribution is achieved by a static load distribution made prior to the execution of the parallel operation. To achieve a balanced distribution in the heterogeneous machine the relative amount of data assigned to each processor, l_i, is a function of its processing capacity compared to the entire machine:

$$l_i = S_i / \sum_{k=1}^{p} S_k \tag{3}$$

For matrices, due to block indivisibility it is not always possible to ensure an optimal load balancing, however, the scheduler computes the optimal solution for a given network [5]. The processor placement on the grid is also done in order to achieve a balanced distribution.

4 Parallel Implementation of the Active Contour Algorithm

An active contour is defined as an energy minimising curve subjected to the action of internal forces and influenced by image forces which move the contour to the relevant features in the image such as lines and edges [19].

Active contours can be used in a diversity of feature extraction operations in images, such as detection of lines and edges, detection of subjective contours, track analysis in a sequence of images or correspondence analysis in stereo images.
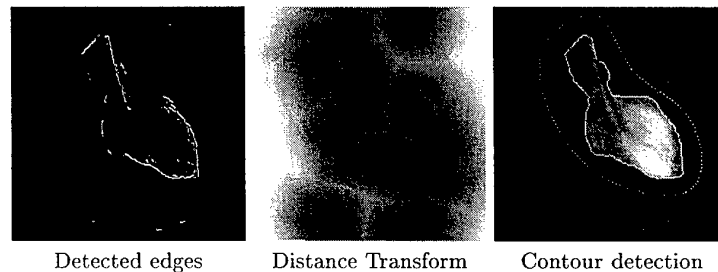

| Detected edges | Distance Transform | Contour detection |

Fig. 4. Application of the active contour algorithm in an angiocardiographic image

Figure 4 (rightmost image) shows the contour detection over the original image of 64 KB. From an initial position (arbitrary or interactively defined), using an iterative process, the contour moves in order to minimise its energy. The final position corresponds to a local minimum of the defined energy function. In this position, the forces applied to the contour are mutually cancelled, such that the contour does not move. The energy function was computed based on the edge detection map (leftmost image) and the distance transform map (middle image). The quality of the detection depends on these two images. Different energy functions can be used [24], however, not all are suitable for every application.

The contour points distant from the edges are pushed in their direction by the distance transform. The points near edges are influenced by the edge map energy which locally refines the detection.

Figure 5 shows the tasks required to apply the active contour algorithm. The computation methodology is to sequentially execute each parallelised task, choosing the processors grid that minimises the individual processing time and consequently the overall time.
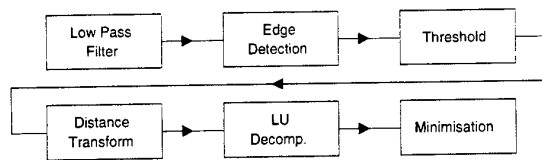
Fig. 5. Active contour algorithm decomposed in indivisible tasks

The image operators have been discussed in another paper [4]. Therefore, only the parallelisation of the LU factorisation routine is considered here.

4.1 LU Factorisation Algorithm

The LU factorisation algorithm is applied in order to solve directly the system of equations resulting from the active contour internal forces: elasticity and flexibility. The implementation follows the right-looking variant of the algorithm proposed in [12]. However, adaptations where made at the load distribution level in order to obtain a balanced load for heterogeneous machines. Figure 6 (left) shows the load distribution obtained in a heterogeneous virtual machine.
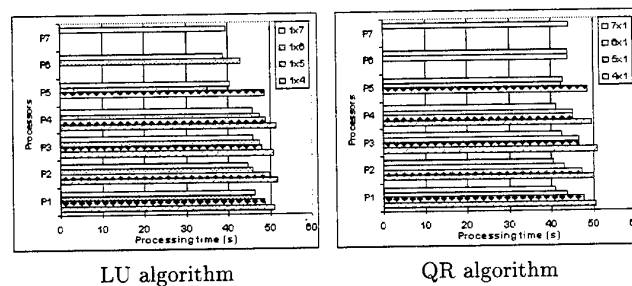

LU algorithm   QR algorithm

Fig. 6. LU and QR load distribution for a matrix size of 1800 and 1600 respectively for the machine M={244, 244, 161, 161, 60, 50, 49} Mflops processors

For processor grids (1,4) and (1,5) a very good load balancing is achieved. For the other grids the three slower processors took approximately 15% less time than the other ones, due to the block indivisibility. The algorithm requires a significant number of communication points which results in a not very scalable algorithm as shown in figure 7 (left).

The scalability analysis was made in a homogeneous machine in order to reduce the influence of load imbalances.
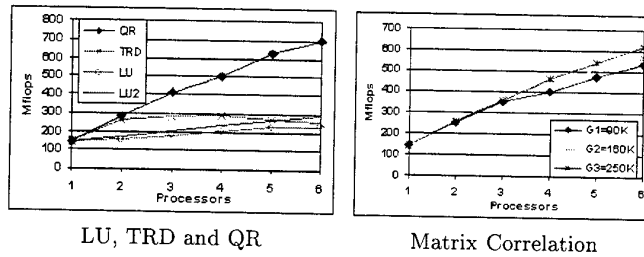
LU, TRD and QR   Matrix Correlation

Fig. 7. Isogranularity curves for a 6 processor homogeneous machine connected by a 10 Mbit/s Ethernet; 160 K elements for TRD, LU and QR and 250K for LU2

5 Parallel Implementation of the Modal Matching Algorithm

This high level image processing algorithm [26] is applied for the tracking of deformable objects over a sequence of images. Figure 8 shows the application of the algorithm. It is based on finite element analysis requiring the computation of eigenvectors of symmetric matrices. The aim is to obtain correspondences between object points of image i and $i+n$. The algorithm is divided into eigenvector computation and matrix correlation. The eigenvector computation is subdivided into three operations: tridiagonalisation, correspondent orthogonal matrix and QR iteration. The parallelisation is then realised by the individual parallelisation of each operation. Data is redistributed if the processor grid changes between operations.

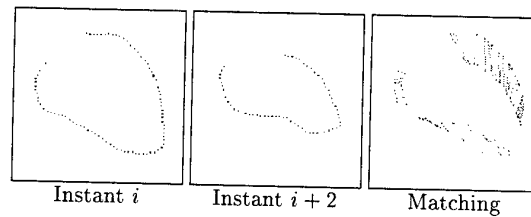Instant i Instant $i+2$ Matching

Fig. 8. Application of the modal analysis algorithm to a sequence of the heart beaten

5.1 Tridiagonal Reduction and Orthogonal Matrix Computation

Tridiagonal reduction is the first algorithm applied to the symmetric matrix in order to obtain the eigenvectors. The algorithm output is a tridiagonal matrix T so that:

$$A = Q^T T Q \tag{4}$$

The matrix T replaces A in memory. As shown in figure 9 the best grid is a row of processors. Details of the algorithm can be found in [8].

The matrix elements of T, apart from the tridiagonal positions, store the data required for the second step of the eigenvector algorithm, i.e. the computation of Q.

If the order of computation of the tridiagonal reduction was followed, an $O(n^4)$ algorithm would be obtained, corresponding to a matrix by matrix product in each step; $n-2$ steps for a matrix of size (n, n). However, the computation can be efficiently organised as described in [22] for a sequential algorithm, obtaining a scalable operation for the virtual machine. Figure 9 shows that the best grid is a row of processors.

5.2 The Symmetric QR Iteration

The QR iteration is the last operation for the eigenvector computation. The aim is to obtain from the tridiagonal matrix T one diagonal Λ where the elements are the eigenvalues of A:

$$T = G^T \Lambda G \qquad (5)$$

The matrix G is then used to compute the eigenvectors Q' of A:

$$Q' = QG^T \qquad (6)$$

Matrix G^T is obtained by iterating and updating it with the Givens rotations [15]. To obtain Q' a matrix by matrix product would be required. However, the operations can be organised in order to update Q' in each iteration avoiding the last matrix product. In the update only two columns of Q' are updated. Based on this fact a scalable operation was implemented by allowing the redistribution of data. The optimal data distribution is blocks of rows so that any given row is completely allocated to a given processor, avoiding communications between processors for the update of Q'. The parallelisation implemented keeps the $O(n^2)$ chase operation in one processor which computes all rotations for an iteration, and distributes them over a column of processors. Then all processors update their rows, the $O(n^3)$ part, in parallel without communications. This strategy has a huge impact in the scalability of the QR iteration as shown by the isogranularity curve in figure 7. A good load balancing is also achieved for a heterogeneous machine as shown in figure 6.

The ideal grid for QR iteration is the opposite (column vs. row) of the ones for tridiagonal and orthogonal matrix computation. This is the reason for considering indivisible operations and allowing redistribution of data between them to adapt the parallel machine to each operation.

5.3 Matrix Correlation

After QR iteration has been computed for the objects in both images the eigenvectors are ordered in decreasing order of magnitude of the correspondent eigenvalue. The correlation operation measures the similarity between the eigenvectors

of both objects. The behaviour of the processing time function shown in Figure 9 is different from the other operations. The best grid is either a row or a column of processors. The parallel algorithm is scalable as shown in figure 7.
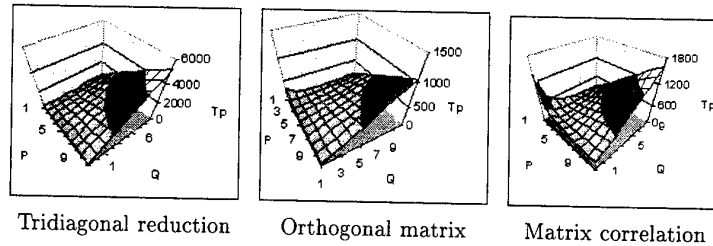

Tridiagonal reduction   Orthogonal matrix   Matrix correlation

Fig. 9. Estimated processing time for a 6 processor homogeneous machine connected by a 10 Mbit/s Ethernet

6 Results

Results are presented for machine M1 composed by 6 homogeneous processors of 141 Mflop each, M2={244, 244, 161, 161, 60, 50, 49} Mflop and M3={161, 161, 112, 80} Mflop processors. M1 is connected by 10 Mbit/s Ethernet, and M2 and M3 by a 100 Mbit/s one. The performance metrics used to evaluate the parallel application is, first, the runtime, and second the speedup achieved. To have a fair comparison in terms of speedup, one defines the Equivalent Machine Number ($EMN(p)$) which considers the power available instead of the number of machines that, for a heterogeneous environment, is an ambiguous information. Equation 7 defines $EMN(p)$ and heterogeneous efficiency E_H, for p processors used, where S_1 is the computational capacity of the processor that executed the serial code, also called the master processor.

$$EMN(p) = \frac{\sum_{i=1}^{p} S_i}{S_1}, \qquad E_H = \frac{Speedup}{EMN(p)} \tag{7}$$

For the machine M3 $EMN(4) = 3.19$, i.e. using 4 processors of the heterogeneous machine is equivalent to 3.19 processors identical to the master processor if it is the 161 Mflop one.

The right table of figure 10 presents results for the parallel active contour algorithm in the M3 machine for an image of 64 KB (figure 4) and for a 256 KB one (the left picture in figure 10). The time T_1 represents the processing time of the serial code in the master processor and T_P the parallel processing time in the virtual machine. The number of processors selected in each step of the algorithm changes in order to minimise the processing time.
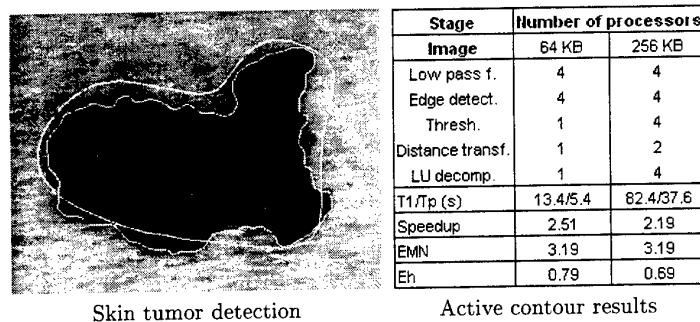
Stage	Number of processors	
Image	64 KB	256 KB
Low pass f.	4	4
Edge detect.	4	4
Thresh.	1	4
Distance transf.	1	2
LU decomp.	1	4
T1/Tp (s)	13.4/5.4	82.4/37.6
Speedup	2.51	2.19
EMN	3.19	3.19
Eh	0.79	0.69

Skin tumor detection   Active contour results

Fig. 10. Application results of the active contour algorithm

Stage	Number of processors used							GRID
(n)	400	600	800	1000	1200	1400	1600	(pxq)
TRD	1	2	3	3	4	4	4	1xq
Q (Orth.)	5	6	6	6	7	7	7	1xq
QR it	5	6	6	6	6	6	6	px1
Speedup	1.0	1.7	2.3	2.6	2.9	3.0	3.1	
EMN	3.6	3.8	3.8	3.8	4.0	4.0	4.0	
Eh	0.28	0.45	0.61	0.68	0.73	0.76	0.78	

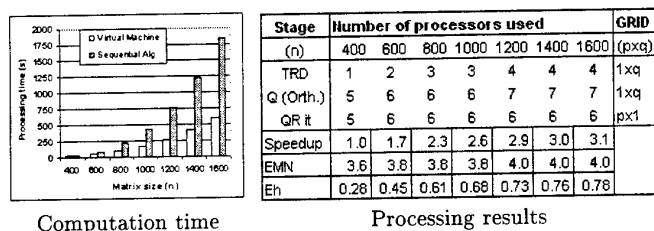Computation time   Processing results

Fig. 11. Eigenvector computation in the M2 machine

Results for of the eigenvector computation are presented in figure 11 for machine M2 due to the wide application of the algorithm. As shown, the heterogeneous efficiency is near 80% for matrices with more than 1400^2 elements. However, the first metric is processing time which is reduced for matrices larger than 400^2 elements.

To show the importance of the parallel processing system, results for the modal analysis algorithm are presented for the homogeneous machine M1, figure 12. The left chart compares the computation time of the virtual machine VM when the optimal number of processors is selected, as indicated in the processing results table, against the processing time when the same number of processors are used for all stages of the algorithm. The minimum time is obtained with 4 processors, however, it is higher than the time obtained for VM.

7 Conclusions

A operation based parallel image processing system for a cluster of personal computers was presented. The main objective is that the user of a computationally demanding application may benefit from the computational power distributed over the network, while keeping other active users undisturbed.

This goal can be achieved in a transparent manner for the user, once the modules of his/her application are correctly parallelised for the target network
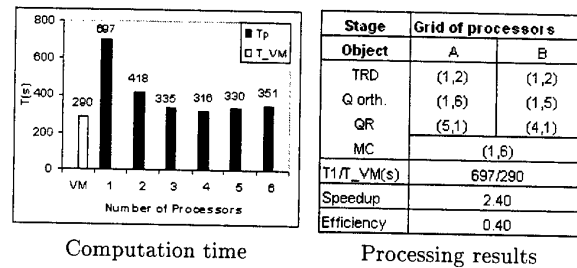
Stage	Grid of processors	
Object	A	B
TRD	(1,2)	(1,2)
Q orth.	(1,6)	(1,5)
QR	(5,1)	(4,1)
MC	(1,6)	
T1/T_VM(s)	697/290	
Speedup	2.40	
Efficiency	0.40	

Computation time   Processing results

Fig. 12. Modal analysis in the homogeneous machine M1

and the performance of the machines in the network is known. The application, before initiating a parallel module, determines the best available computer composition for a parallel virtual computer to execute it, and then launches the module, achieving the best response time possible in the actual network conditions.

Practical tests were conducted both on homogeneous and heterogeneous networks. In both cases the theoretically optimal computer grid was confirmed by the measured performance. A balanced load was achieved in both machines. The machine scalability depends essentially on the communication requirements of the operations. For QR iteration and matrix correlation the system is scalable, however, it is not for the tridiagonal reduction.

Other generic modules will be parallelised and tested, so that an ever increasing number of image analysis methods may be assembled from them. Application domains other than image analysis may also benefit from the proposed methodology.

References

1. A. Alves, L. Silva, J. Carreira, and J. Silva. Wpvm: Parallel computing for the people. In *HPCN'95 High Performance Computing and Network Conference*, Milan (http://dsg.dei.uc.pt/wpvm), 1995. Springer-Verlag.
2. T. Anderson, D. Culler, D. Patterson, and The NOW Team. A case for now (network of workstations). *IEEE Micro*, February 1995.
3. D. Bader, J. JáJá, D. Harwood, and L. S. Davis. Parallel algorithms for image enhanced and segmentation by region growing with an experimental study. Technical Report UMCP-CSD:CS-TR-3449, University of Maryland, May 1995.
4. J. Barbosa and A.J. Padilha. Algorithm-dependent method to determine the optimal number of computers in parallel virtual machines. In *VECPAR'98, 3rd International Meeting on Vector and Parallel Processing (Systems and Applications)*, volume 1573, Porto, 1998. Springer-Verlag.
5. J. Barbosa, J. Tavares, and A. J. Padilha. Linear algebra algorithms in a heterogeneous cluster of personal computers. In *Accepted for presentation in the 9th Heterogeneous Computing Workshop*, Cancun, Mexico, 2000. IEEE Computer Society.

6. J. Canny. A computational approach to edge detection. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 8(6), November 1986.

7. J. Choi, J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. The design and implementation of the scalapack lu, qr, and cholesky factorization routines. *Scientific Programming*, 5:173–184, 1996.

8. J. Choi, J. Dongarra, and D. Walker. The design of parallel dense linear software library: Reduction to hessenberg, tridiagonal and bidiagonal form. Technical Report LAPACK Working Note 92, University of Tennessee, Knoxville, January 1995.

9. D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *4 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, 1993.

10. J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

11. H. Derin and C.-S. Won. A parallel image segmentation algorithm using relaxation with varying neighborhoods and its mapping to array processors. *Computer Vision, Graphics, and Image Processing*, (40):54–78, 1987.

12. J. Dongarra, Sven Hammarling, and David W. Walker. Key concepts for parallel out-of-core lu factorization. Technical Report CS-96-324, LAPACK Working Note 110, University of Tennessee Computer Science, Knoxville, April 1996.

13. J. Dongarra and D. Walker. The design of linear algebra libraries for high performance computers. Technical Report LAPACK Working Note 58, University of Tennessee, Knoxville, June 1993.

14. R. Geijn and J. Watts. Summa: Scalable universal matrix multiplication algorithm. Technical Report CS-95-286, University of Tennessee, Knoxville, 1995.

15. Gene Golub. *Matrix Computations*. The Johns Hopkins University Press, 1996.

16. J.F. JáJá and K.W. Ryu. The block distributed memory model. Technical Report CS-TR-3207, University of Maryland, January 1994.

17. Dan Judd, Nalini K. Ratha, Philip K. McKinley, John Weng, and Anil K. Jain. Parallel implementation of vision algorithms on workstation clusters. In *Proceedings of the International Conference on Pattern Recognition*, pages 317–321, Jerusalem, 1994. IEEE Press.

18. Zoltan Juhasz and Danny Crookes. A pvm implementation of a portable parallel image processing library. In *EuroPVM 96*, volume 1156, pages 188–196, Munich, 1996. Springer-Verlag.

19. Michael Kass, Andrew Witkin, and Demetri Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, pages 321–331, 1988.

20. Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - a hunter of idle workstations. In *Proc. IEEE 8th International Conference on Distributed Computing Systems*, pages 104–111, Los Alamitos, California, 1988. IEEE CS Press.

21. M. Luckenhaus and W. Eckstein. A thread concept for automatic task parallelization in image analysis. In *Proceedings of Parallel and Distributed Methods for Image Processing II*, volume 3452, pages 34–44, San Diego, California, 1998. SPIE.

22. W.H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1997.

23. E. S. Quintana, G. Quintana, X. Sun, and R. Geijn. Efficient matrix inversion via gauss-jordan elimination and its parallelization. Technical Report CS-TR-98-19, University of Texas, Austin, September 1998.

24. L. P. Reis, J. Barbosa, and J. M. Sá. Active contours: Theory and applications. In *RECPAD96 - 8th Portuguese Conference on Pattern Recognition*, Guimarães, Portugal, 1996.

25. C. Seitz. Myrinet - a gigabit per second local-area network. *IEEE Micro*, February 1995.

26. L. Shapiro and J. M. Brady. Feature-based correspondence: an eigenvector approach. *Butterworth-Heinemann Lda*, 10(5), June 1992.

27. J. Shen and S. Castan. An optimal linear operator for step edge detection. *CVGIP: Graphical Models and Image Processing*, 54(2):112–133, March 1992.

28. H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller Jr., H. E. Smalley Jr., and S. D. Smith. Pasm: A partitionable simd/mimd system for image processing and pattern recognition. *IEEE Transactions on Computers*, C-30(12), December 1981.

29. S. M. Smith and J. M. Brady. Asset-2: Real-time motion segmentation and shape tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(8):814–820, 1995.

30. L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

Improving the Performance of Heterogeneous DSMs via Multithreading

Renato J. O. Figueiredo, Jeffrey P. Bradford and José A. B. Fortes*

School of ECE, Purdue University
West Lafayette, IN 47907
{figueire,jbradfor,fortes}@ecn.purdue.edu

Abstract

This paper analyzes the impact of hardware multithreading support on the performance of distributed shared-memory (DSM) multiprocessors built out of heterogeneous, single-chip computing nodes. Area-efficiency arguments motivate a heterogeneous, hierarchical organization (HDSM) consisting of few processors with extensive support for instruction-level parallelism and large caches, and a larger number of simpler processors with smaller caches for efficient execution of thread-parallel code. Such heterogeneous machine relies on the execution of multiple threads per processor to deliver high performance for unmodified applications. This paper quantitatively studies the performance of HDSMs for software-based and hardware-multithreaded scenarios. The simulation-based experiments in this paper consider a 16-node multiprocessor, six homogeneous shared-memory benchmarks from the SPLASH-2 suite, and a decision-support application (C4.5). Simulation results show that a hardware-based, block-multithreaded HDSM configuration outperforms a software-multithreaded counterpart, on average, by 13%.

1 Introduction

Continuing technological advances in VLSI manufacturing are predicted to bring about billion-transistor chips in the next decade [15]. Such large transistor budget allows for the implementation of high-performance uniprocessors [12] that aggressively exploit instruction-level parallelism (ILP), as well as chip-multiprocessors [8] that can efficiently execute explicitly parallel tasks.

Large multiprocessor configurations of the future will be able to use such high-performance components as commodity building blocks in their design. Previous work [6] has shown that combining nodes of different processor and memory characteristics into a heterogeneous distributed shared-memory (**HDSM**) multiprocessor leads to area-efficient designs.

An HDSM combines few high-performance processors and memories with a larger number of simpler processors and smaller memories to form a hierarchical, heterogeneous system [1] capable of fast execution of both sequential and parallel codes. Figure 1 depicts the organization of an HDSM.
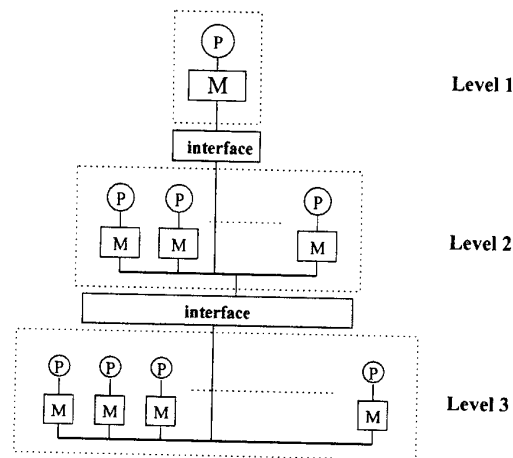

Fig. 1. *HDSM: processor-and-memory hierarchical organization. Processors and memories are drawn such that processor performance and memory capacity are proportional to their area in the figure.*

The proposed heterogeneous DSM machines rely on the execution of multiple threads per processor to deliver high performance for unmodified, homogeneous applications. Previous work has studied the performance of HDSMs assuming a software multi-tasking model based on voluntary context switches. This model is valid for current commodity microprocessors that do not provide hardware mechanisms to implement fine-grain multithreading. However, hardware multithreaded microarchitectures are currently being used in commercial processors [16] and considered in the implementation of future-generation high-performance microprocessors [4].

This paper extends the performance studies of HDSMs reported in [6] by quantitatively analyzing the impact of hardware multithreading on their performance. This paper also complements previous work by employing a simulation model that explicitly accounts for heterogeneity of processor performance due to ILP.

The quantitative analysis is performed via simulation of parallel benchmarks from the SPLASH-2 suite [18] and of a hand-paralellized decision-support application (C4.5 [13]). Benchmarks are simulated individually to study single-program parallel speedup. All benchmarks are programmed with single-program.

multiple-data (SPMD) extensions to the C language.

A modified version of the RSIM [10] multiprocessor simulator is used in the experiments. The original RSIM simulator models DSM machines built out of homogeneous ILP processors, with no hardware support for multithreading. It has been modified for the performance analysis shown in this paper to model heterogeneity of ILP processors and caches, and to model hardware support for multithreading.

This paper is organized as follows. Section 3 describes the heterogeneous DSM machine model studied in this paper. Section 3 presents the experimental methodology used in the performance study. Section 4 presents experimental results and data analyses. Section 5 concludes this paper.

2 Machine model

2.1 Heterogeneous DSMs

HDSM machines differ from conventional distributed shared-memory multiprocessors in that processors, memories and networks of HDSMs may be heterogeneous. In this paper, processor heterogeneity is modeled in terms of degree of support for ILP. Heterogeneity in the memory subsystem is modeled in terms of L1 and L2 cache sizes and access times. Heterogeneity of the network subsystem is not modeled in this paper.

The heterogeneity of processors and caches is motivated by area/parallelism tradeoffs in the design of future-generation microprocessors: the system consists of a combination of few, aggressive uniprocessors with large caches and many simpler processors with smaller individual caches. The former processors devote large numbers of transistors to deliver high performance for sequential codes, while the latter processors have smaller silicon area requirements and deliver high performance for parallel codes.

The area/parallelism argument that motivates the design of HDSMs is based on the use of area-efficient simple processors for execution of parallel codes, and aggressive uniprocessors for execution of sequential codes. For highly parallel tasks, the high-performance uniprocessors can also be assigned to parallel computation.

Previous work has shown that a software-based assignment of multiple threads to the high-performance ILP uniprocessors of an HDSM yields performance improvements for memory- and cpu-intensive programs [6]. Context switches in software multi-tasking occur infrequently, and have large execution time overheads. Such coarse-grain model limits the potential for overlapping high-latency shared-memory DSM operations.

Research on multi-threaded processors has shown that aggressive ILP uniprocessors can be enhanced to support multiple threads with small increases in chip area requirements [4]. The implementation of hardware multi-threading extensions into the aggressive ILP processors of an HDSM can increase overall system performance by increasing the potential for overlapping of shared-memory accesses. To investigate the performance of such enhanced system, the

high-performance processors of the HDSM machine modeled in this paper have hardware support for block-multithreading.

2.2 HDSM multiprocessor configuration

The HDSM multiprocessor under study consists of sixteen nodes. Each node contains a single processor, L1 and L2 data caches, main memory and a remote access device (**RAD**) with network interface and coherence controller. The nodes are interconnected by a 2-D mesh. Cache coherence is maintained via a directory controller that implements the MESI [11] protocol. The release consistency [7] memory model is assumed in this study. Figure 2 depicts the machine model.
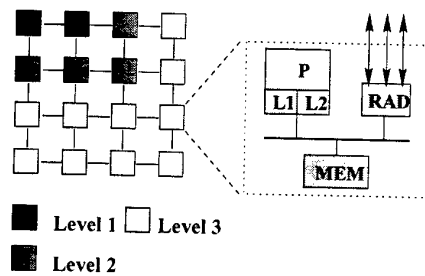

Fig. 2. *HDSM model: each heterogeneous node has a single processor (**P**), two levels of data cache (**L1,L2**), main memory (**MEM**) and a remote access device (**RAD**), all connected by a memory bus. Nodes are interconnected via a mesh network.*

Heterogeneity is present in both the processor and memory subsystems of the HDSM machine. Processor heterogeneity is modeled in terms of the size of hardware structures dedicated to ILP exploitation. The heterogeneous ILP parameters investigated in this paper are issue rate, instruction window size, number of arithmetic (ALU), floating-point (FPU) and address units, and maximum number of outstanding cache misses (MSHRs [9]). Heterogeneity in the memory subsystem is modeled in terms of the size and speed of caches.

The HDSMs under study have three levels, with 2, 4 and 10 nodes in levels 1, 2 and 3, respectively. The machine is configured as a processor-and-memory hierarchy [1]; the number of processing elements increases from top to bottom levels of the hierarchy, while cache sizes and the performance of processors and cache memories decrease from top to bottom levels. Table 1 shows the processor and memory configurations assumed for each machine level.

The inter-processor network is assumed to be homogeneous. This assumption is conservative in accounting for inter-processor communication latencies. Given the predicted integration level of next-generation microprocessors, it is conceivable that HDSM levels built out of simple processors be integrated into

	level **i=1**	level **i=2**	level **i=3**
number of processors(i)	2	4	10
issue witdh(i)	8	4	1
instruction window size(i)	128	64	8
number of ALU/FPU/address units(i)	4	2	1
number of MSHRs(i)	12	8	4
L1 cache size(i)	32KB	16KB	8KB
L2 cache size(i)	1MB	256KB	64KB
L2 cache miss detection latency(i)	10	5	3
L2 cache hit latency(i)	25	13	8

Table 1. *3-level, 16-processor heterogeneous machine configuration. L2 cache miss detection and hit latencies are shown in terms of clock cycles.*

single-chip multiprocessors [8, 6]. Such configuration would allow smaller intra-level latencies than those assumed in the machine model.

2.3 Heterogeneous node configurations

The configuration of the level-3 processor is based on a simple out-of-order microprocessor pipeline that issues one instruction per cycle. The level-2 configuration is based on current high-performance, out-of-order microprocessor designs [5]. The high-performance level-1 processor is based on predicted configurations of future-generation ILP microprocessors [3, 14].

The cache sizes of the level-1 processor are dimensioned so that the L1 and L2 data caches are large enough to hold the primary and secondary working sets, respectively, of the SPLASH-2 benchmarks [18]. Cache sizes of lower-level processors are scaled down (with respect to the adjacent upper level) by factors of 2 (L1 cache) and 4 (L2 cache).

The L1 cache access times are assumed to be a single processor cycle for all processor configurations: it is assumed that clock cycles are the same for all processors and that the level-1 caches are designed to match the pipeline clock. The L2 cache tag and data access times are modeled after the analytical cache access time model described in [17], assuming a $0.18\mu m$ technology [14].

The remaining processor and memory simulation parameters are homogeneous across HDSM nodes and are set to the default values of the original RSIM simulator.

2.4 Programming model

This paper considers the execution of homogeneous parallel applications on HDSMs. These programs are written in the single-program, multiple data (SP-MD) model. The homogeneous programs are mapped onto heterogeneous re-

sources without source code modifications via static thread-to-processor assignment schemes. The next subsection details the two assignment schemes studied in this paper.

2.5 Multi-threading model

In this paper, two policies are considered in the assignment of threads to heterogeneous processors. In the virtual-processor policy, both software and hardware support for multithreading are studied.

1. *Single-thread:* one thread is assigned to each processor in the system.
2. *Virtual-processor:* in this scheme, a processor P_i is assigned $VP(i)$ threads, where $VP(i)$ is the ratio between P_i's performance and the slowest processor in the system. This ratio is obtained from the uniprocessor simulation results summarized in Figure 3 (benchmarks that require power-of-two number of processors are assigned 5, 3, and 1 threads to processors in levels 1, 2, and 3, respectively). There are two different multithreading scenarios studied under this assignment policy:
   (a) *Software multithreading:* in this scenario, thread context switches are triggered only by failed synchronizations on locks and barriers. To implement this switching criterion, the RSIM synchronization library has been modified to include a voluntary context-switch call in the spin-waiting loop of the synchronization operations. The software context-switching overhead is modeled in the simulator by forcing the switching processors to be idle for a configurable number of clock cycles. The context switching overhead in this scenario is 800 processor cycles.
   (b) *Hardware multithreading:* in this scenario, hardware support for block-multithreading [2] is available in the HDSM level-1 and level-2 processors. Thread context switches are triggered by the following criteria (in addition to failed synchronization): when L2 cache misses occur, when the number of cycles without any instruction graduation exceeds the threshold T_{grad}, and when the total number of cycles without any thread context switch exceeds the threshold T_{switch}. In this paper, T_{grad} and T_{switch} are set to 20 and 10000 processor cycles, respectively. The context switching overhead is this scenario is set to 4 processor cycles. In addition, threads are guaranteed not to be context-switched for a minimum run length of 4 cycles.

3 Experimental methodology

3.1 Benchmarks

The set of benchmarks used in this paper includes six programs from the SPLASH-2 [18] suite and a parallelized version of the decision-support database program C4.5 [13].

The programs (and respective data sets) studied in this paper are *C4.5* (adult dataset with unknowns removed and a minimum node size of 100), *FFT* (16K points), *FMM* (4096 particles), *LU* (256x256 matrix), *Ocean* (258x258 ocean), *Radix* (512K integers) and *Water* (512 molecules). All benchmarks are compiled with Sun Microsystem's WorkShop C compiler version 4.2 and optimization level **-x04**.

3.2 Simulation environment

The simulation environment is based on a modified version of the RSIM simulator [10] that models a release-consistent DSM machine connected by a 2-D mesh, with uniprocessor heterogeneous nodes with support for block-multithreading.

4 Experimental results

In this section, the performance of HDSMs is analyzed for the thread assignment schemes described in Section 3. Initially, the relative performance of the individual heterogeneous processors is discussed. Subsequently, the impact of multithreading on HDSM performance is analyzed.

4.1 Impact of ILP heterogeneity on single-node performance

Figure 3 shows the performances of the heterogeneous processors and caches in terms of speedups with respect to a base (level-3) processor. The level-2 and level-1 processors outperform the single-issue level-3 processor, on average, by 277% and 396%, respectively. Since clock speeds are assumed to be the same for all processors, the performance differences between the heterogeneous processors are due to instruction-level parallelism and cache sizes only.

Figure 3 shows that an eight-fold increase in issue rate and a sixteen-fold increase in L2 cache yield an average four-fold performance improvement of the level-1 processor over the simple level-3 processor. This result is consistent with the area-efficiency analysis based on a case study of Alpha microprocessors presented in [6]. The increase in chip area necessary to implement larger caches and structures devoted to the extraction of ILP yields sub-linear gains in performance under the assumption of same fabrication technology (and clock cycle).

4.2 Parallel speedup analysis

Figure 4 shows the speedups of the 16-node HDSM with respect to the base (level-3) processor for the three different assignment scenarios described in Section 3. In the virtual-processor assignment, 4, 2 and 1 threads are assigned to level-1, level-2 and level-3 processors, respectively (except for benchmarks that require power-of-two processors, where 5, 3 and 1 threads are assigned to processors of levels 1, 2 and 3). The simulation results show that the virtual-processor
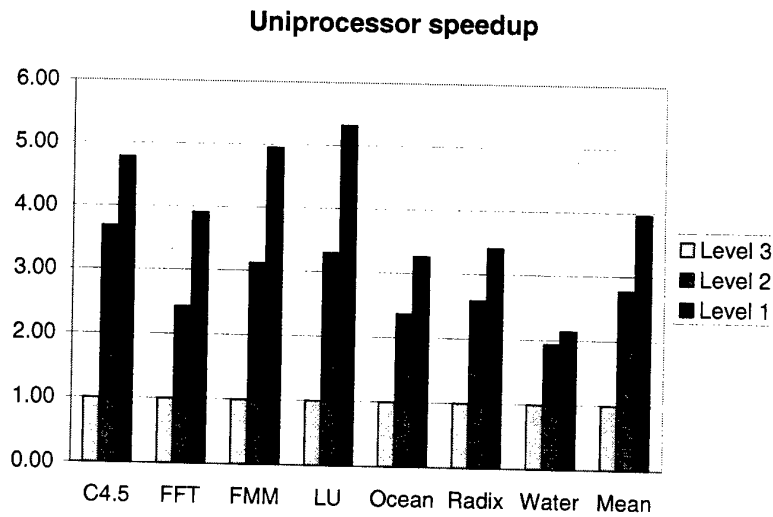
Uniprocessor speedup

Fig. 3. *Simulated uniprocessor speedups (with respect to level-3 processor) of the heterogeneous configurations. shown in Table 1.*

assignment significantly outperforms the single-thread assignment under both studied multithreading models. The average virtual-processor speedups are 28% and 45% for the software and hardware multithreaded schemes, respectively.

The hardware multithreading model outperforms the software model for all benchmarks except *Radix*: the largest performance improvement is observed in *FFT* (21.6%), followed by *C4.5* (19.6%), *FMM* (16.5%). *Ocean* (15.4%). *LU* (9.4%) and *Water* (7.0%). For *Radix*, the hardware multithreading model performs as well as the hardware model. These results can be explained with a closer analysis of the execution time in the level-1 processor.

Figures 5. 6 and 7 show a breakdown of the execution time in one of the level-1 processors into three components: busy, stalled on memory accesses and stalled on synchronization (locks and barriers) for the three assignment scenarios of Figure 4.

In the single-thread case (Figure 5), the high-performance level-1 processor spends most of its execution in synchronization points. Since this assignment does not account for heterogeneity in processor performance. the level-1 processor is often waiting to synchronize with lower-level (slower) processors to proceed with computation.

In the software multithread case (Figure 6), the level-1 processor spends less time in synchronization relative to actual computation. The load-balancing property of the virtual-processor scheme allows the level-1 processor to perform more computation before attempting to synchronize with lower-level processors. and
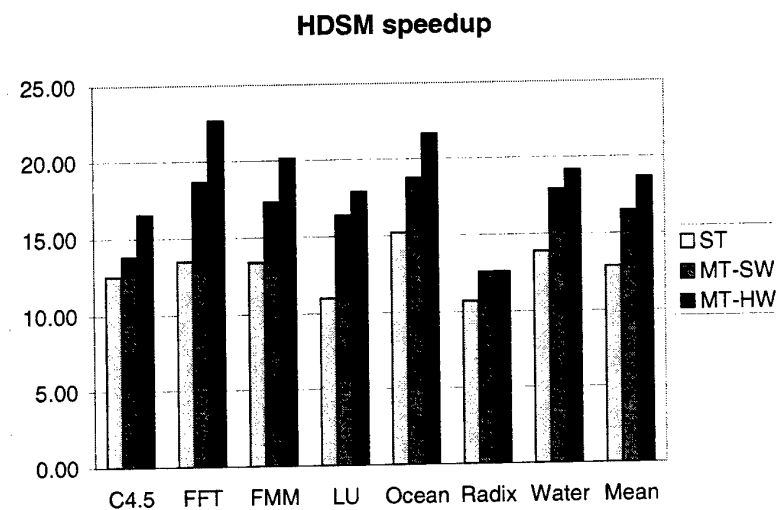
HDSM speedup

Fig. 4. *Simulated HDSM speedups (with respect to level-3 processor) for single-thread and virtual-processor assignments (software and hardware multithreading models).*
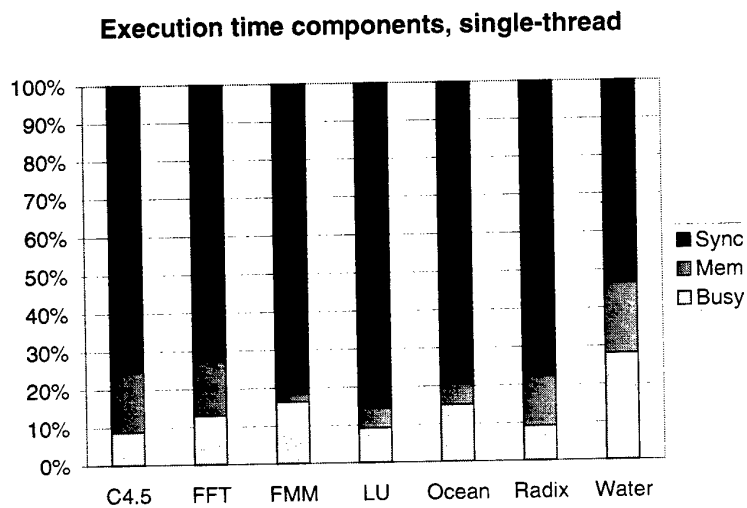
Execution time components, single-thread

Fig. 5. *Relative contributions of busy, memory and synchronization to total execution time of a level-1 processor under the single-thread assignment.*
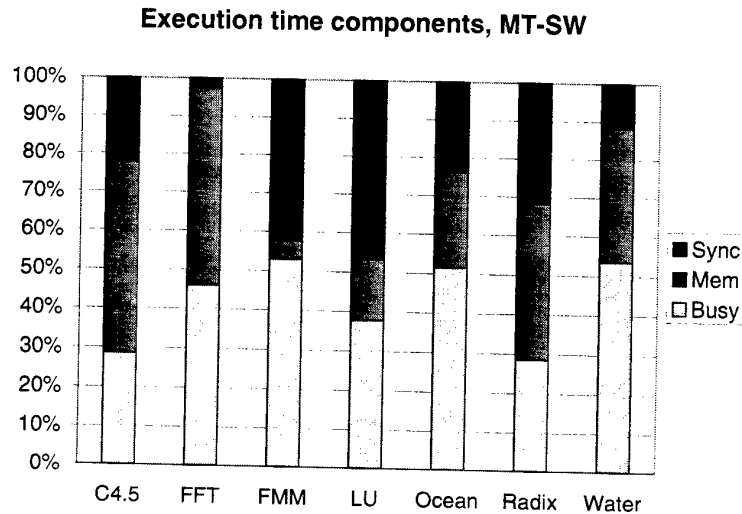
Execution time components, MT-SW

Fig. 6. *Relative contributions of busy, memory and synchronization to total execution time of a level-1 processor under the virtual-processor, software multithreading assignment.*

hence the synchronization component is reduced significantly. Since the processor spends less time in synchronization points, the (relative) busy and memory components increase.

A comparison of the multithread cases (Figures 6 and 7, respectively) shows that, for all benchmarks (in particular, *C4.5* and *FFT*), the relative memory access component gets reduced when hardware support is present. This is explained by the ability of hardware multithreading to hide memory latencies by overlapping memory accesses from distinct threads. The improved memory behavior is reflected in increased processor usage (busy component) and, ultimately, in better performance over the software scheme as shown in Figure 4.

For *Radix*, the hardware scheme fails to deliver better performance for the following reason. In *Radix*, the increased frequency of context switches causes interference in the level-1 cache, increasing the worst-case L1 miss rate in processor 0 (HDSM level 1) from 9.7% to 15.1%.

5 Conclusions

A heterogeneous, hierarchical organization of processor and memory resources of a DSM allows efficient execution of codes with various degrees of parallelism. This organization also delivers high-performance for unmodified, homogeneous shared-memory parallel programs that exhibit a single degree of parallelism.
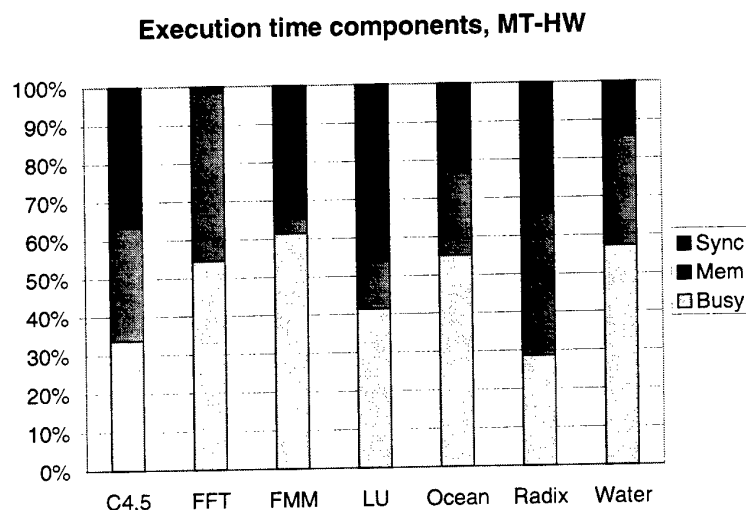
Execution time components, MT-HW

Fig. 7. *Relative contributions of busy, memory and synchronization to total execution time of a level-1 processor under the virtual-processor, hardware multithreading assignment.*

Support for the execution of multiple threads in the high-performance processors of a heterogeneous DSM is key to delivering high performance for such homogenous parallel applications. This paper shows that the virtual-processor assignment of threads to nodes that are heterogeneous only with respect to ILP hardware and cache sizes improves the average performance of HDSMs by up to 45%, when compared to a single-thread assignment policy.

This paper also shows that hardware support for hardware block multithreading in the high-performance upper-level processors is desirable for an HDSM organization. A simulation analysis shows that hardware multi-threading improves the performance of virtually-assigned homogeneous applications in HDSMs by as much as 21% (13% on average) over a software-based context-switching scheme.

A detailed analysis of the execution in the multithreaded upper-level processors shows that, while the virtual-processor thread assignment mechanism is able to improve load balancing. the hardware multithreading solution is particularly effective in overlapping high-latency shared-memory accesses and reducing the memory component of the execution time.

References

1. Ben-Miled. Z. and Fortes. J.A.B. A Heterogeneous Hierarchical Solution to Cost-efficient High Performance Computing. *Par. and Dist. Processing Symp..* Oct 1996.

2. Boothe, B. and Ranade, A. Improved Multithreaded Techniques for Hiding Communication Latency in Multiprocessors. In *Proc. 19th International Symposium on Computer Architecture*, pages 214–223, 1992.

3. Chrysos, George Z. and Emer, Joel S. Memory Dependence Prediction Using Store Sets. In *Proc. 25th International Symposium on Computer Architecture*, pages 142–153. 1998.

4. Eggers, S. J., Emer, J. S., Levy, H. M., Lo, J. L., Stamm, R. L., and Tullsen, D. M. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, pages 12–19, Sep. 1997.

5. B. A. Gieseke et al. A 600MHz Superscalar RISC Microprocessor with Out-of-Order Execution. In *International Solid State Circuits Conference*, 1997.

6. Figueiredo, R. J. O. and Fortes, J. A. B. Impact of Heterogeneity on DSM Performance. In *Proc. 6th International Symposium on High-Performance Computer Architecture*, pages 26–35, Jan 2000.

7. Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta. A., and Hennessy. J. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *Proc. 17th International Symposium Computer Architecture*, June 1990.

8. Hammond, L., Nayfeh, B. A., and Olukotun, K. A Single-Chip Multiprocessor. *IEEE Computer*, Sep. 1997.

9. Kroft, D. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proc. 8th International Symposium on Computer Architecture*, 1981.

10. Pai, V. S., Ranganathan, P., and Adve, S. V. The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proc. 3rd International Symposium on High-Performance Computer Architecture*, Feb 1997.

11. Papamarcos, M. and Patel, J. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proc. of 11th Annual Int. Symp. on Computer Architecture*, 1984.

12. Patt. Y. N., Patel, S. J., Evers, M., Friendly, D. H., and Stark, J. One Billion Transistors, One Uniprocessor, One Chip. *IEEE Computer*, pages 51–57, Sep. 1997.

13. Quinlan, J. R. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, California, 1993.

14. Reinman, G., Austin, T., and Calder, B. A Scalable Front-end Architecture for Fast Instruction Delivery. In *Proc. 26th International Symposium on Computer Architecture*, pages 234–245, 1999.

15. Semiconductor Industry Association. The National Technology Roadmap for Semiconductors. San Jose, CA, 1997.

16. Storino, S. N., Borkenhagen, J. M., Kalla, R. N., and Kunkel. S. R. A Multi-Threaded 64-bit PowerPC Commercial RISC Processor Design. In *Hot Chips XI*, 1999.

17. Wilton, S.J.E. and Jouppi. N.P. An Enhanced Access and Cycle Time Model for On-Chip Caches. Technical Report WRL Research Report 93/5, Western Research Laboratory, Digital Equipment Corporation, 1993.

18. Woo. S. C. et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual Int. Symp. on Computer Architecture*. July 1995.

Solving the Generalized Sylvester Equation with a Systolic Library

Gloria Martínez[1], Germán Fabregat[1], and Vicente Hernández[2]

[1] Dpt. de Informática, Univ. Jaume I, Campus Riu Sec
E-12071 Castellón, Spain
{martine, fabregat}@inf.uji.es
[2] D.S.I.C., Univ. Politécnica de Valencia, Camino de Vera s/n
E-46071 Valencia, Spain
vhernand@dsic.upv.es

Abstract. The study of the solution of the Generalized Sylvester Equation and other related equations is a good example of the role played by matrix arithmetic in the field of Modern Control Theory. We describe the work performed to develop systolic algorithms for solving this equation, in a fast and effective way. The presented results show that the design methodology used allowed us to propose the use of Systolic Libraries, that is, reusable systolic arrays that can be implemented taking profit of the use of FPGA technology. In this paper we show how it is feasible to solve the Generalized Sylvester Equation using basic modules of Linear Algebra that can be implemented on versatile systolic arrays.

1 Introduction.

The Generalized Sylvester Equation, $AXB + CXD = E$, with $A, C \in R^{m \times m}$, $B, D \in R^{n \times n}$ and $X, E \in R^{m \times n}$, and some simpler derived equations such as the Sylvester[7],[15],[3] Lyapunov [13],[17] and Stein [7],[15] have multiple and important applications in the field of Control Theory [9],[7],[15].

Obtaining the solution of these equations is a suitable problem for the efficient use of parallel algorithms, due to the regular structure of the matrices. However, when real-time constraints apply to the system, the use of dedicated processors, usually implementing systolic algorithms in VLSI is required. We have recently presented several works [10],[12] showing that a modular approach to systolic algorithms is a suitable way of building fast, reconfigurable solutions to be implemented in FPGA devices to obtain cost-effective custom processors to solve different problems.

The starting point is a new design methodology [10] based on the *Kronecker Product* and *Vec-Function* operators. Algorithms obtained this way are easy to parallelize because they consist of combinations of basic, widely studied operations (Solve a triangular equation system, Gaxpy, Saxpy, QR decomposition of a Hessenberg matrix, ...), and the required data flow is well structured to pass from one functional block to another without intermediate storage.

Extending these results, we have compiled in a Systolic Library for Linear Algebra all the basic modules, following the same principle of modular programming that generated other sequential and parallel environments [1],[18]. For the modules of this library [11] to be useful to solve any problem in their application field, two restrictions hold: (1) all the systolic arrays must share a compatible data flow, to allow results from one of them be forwarded to another, and (2) the arrays must be designed to process problems of any size. These two restrictions have been satisfied using dynamic arrays and applying the DBT transformation [14] on the basic operations of the linear algebra.

The application described in this paper is a good example of the use of the Systolic Library. The first step to solve the Generalized Sylvester Equation, following the method proposed by Golub, Nash and Van Loan [4], is transforming the original problem $A'X'B' + C'X'D' = E'$, into $AXB + CXD = E$ using orthogonal similarity transformations on the pencils $A' - \lambda C'$ and $D' - \lambda B'$ to obtain their Generalized Schur Forms (that is, $P_1^T(A - \lambda C)P_2^T = A' - \lambda C'$ and $Q_1^T(D - \lambda B)Q_2^T = D' - \lambda B'$). The coefficient matrices of the resulting equation are in a condensed form. We have worked on the solution for three cases [10]: first, when all of them are triangular (*Triangular Case*). Second, when A is Schur or Hessenberg and the others triangular (*Hessenberg Case*). Third, when both matrices A and D are Schur (*General Case*). The study of the two first cases has made possible the development of the basic arrays; the study of the general case allowed us to prove how the collection of routines obtained were efficient (and sufficient) to solve more general and complex problems.

Section 2 presents the basis of the methodology for developing the algorithms: the definition of Kronecker Product and Vector Function of a matrix. Section 3 describes the main operations to be solved when studying the solution of the Generalized Sylvester Equation in the General Case. Then section 4 shows how to use the library to implement this operation. Finally section 5 concludes and presents the ongoing work.

2   Applying the Methodology of Design.

The methodology used to solve the Generalized Sylvester Equation, described in [10], is based on the definition of the Kronecker Product and Vec-Function of a matrix. The properties of both operators [6] can be applied to simplify the structure of the problem. Concretely, by applying them to the equation $AXB + CXD = E$, the linear equation system $(B^T \otimes A + D^T \otimes C)vec(X) = vec(E)$, shown in figure 1, is obtained[1]. The resulting system, too huge to be of practical implementation, offers a clear representation of the data dependencies and a simple expression of the basic steps required to solve the problem.

The structure, similar to an upper triangular system, suggests the application of the Back Substitution Algorithm to solve the problem. For example, an intuitive and simple method would be to obtain the value of x_n and then update

---

[1] Assuming that the pencil $D - \lambda B$ has lower quasi-triangular structure: this affects only to the order of resolution and helps to visualize the problem.
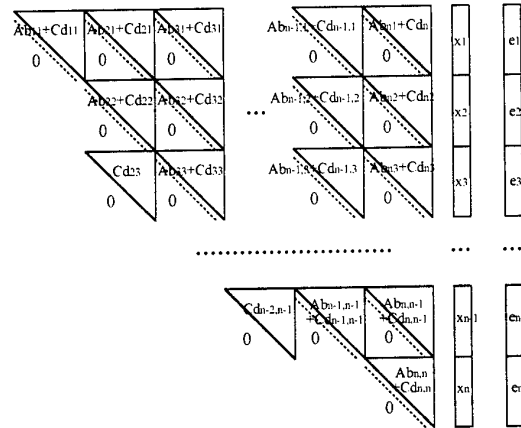
Fig. 1. Linear Equation System obtained by applying the Kronecker Product and the Vec-function to the Triangular Generalized Sylvester Equation.

the values of $e_{n-1}, \ldots, e_1$ as is done for the solution of a triangular system. The resulting procedure is shown in figure 2.

```
Calculate Q: (Ab_ii+Cd_ii)Q is upper triangular;
Solve ((Ab_ii+Cd_ii)Q)(Q^T x_i)=e_i;
w:=(AQ)*(Q^T x_i);
v:=(CQ)*(Q^T x_i);
x_i:=Q*(Q^T x_i);
for j:=i-1 downto 1 do
    Update e_j:=e_j-wb_ij-vd_ij
endfor;
```

Fig. 2. SGH Step: Procedure to obtain x_i, assuming that $d_{i-1,i} = 0$.

But figure 1 also shows that for certain elements (for example x_3), that simple procedure cannot be applied because there are subdiagonal elements of matrix D (d_{23}) that produce subdiagonal blocks in the transformed matrix. It is then necessary to solve at once two columns of matrix X (x_3 and x_2). We will call this new operation **Solve_2**. Figure 3 shows the complete procedure to solve the equation.

In the resulting **SGG Algorithm** all the operations but Solve_2 are basic operations of Linear Algebra and they can be directly performed on the arrays designed in the systolic library described in [11]. In fact, the SGH step is the

```
i:=n;
while (i>0) do
    if (di-1,1=0) then
       SGH step;
       i:=i-1
    else
       Solve_2 (Ab_i-1,i-1+Cd_i-1,i-1 Ab_i,i-1+Cd_j,i-1)(x_i-1) = (e_i-1) ;
                       Cd_i-1,i          Ab_ii+Cd_ii  ( x_i )   ( e_i )

       w1:=A*x_i;

       v1:=C*x_i;

       w2:=A*x_i-1;

       v2:=C*x_i-1;

       for j:=i-1 downto 1 do
          Update e_j:=e_j-(w1b_ij+v1d_ij+w2b_i-1,j+v2d_i-1,j)
       endfor;
       i:=i-2
    endif
endwhile;
```

Fig. 3. The resulting SGG Algorithm.

basic stage of the Algorithm for solving the Hessenberg case [10]. Therefore, to continue with the study of the solution of the General case it is necessary to study this new operation.

3   The SOLVE_2 Operation.

For the efficient implementation of the Solve_2 operation we start by analyzing the structure of its coefficient matrix, $\mathcal{M}$; a possible example, assuming m=4, would be the following

$$\mathcal{M} = \begin{pmatrix} Ab_{i-1,i-1} + Cd_{i-1,i-1} & Ab_{i,i-1} + Cd_{i,i-1} \\ Cd_{i-1,i} & Ab_{ii} + Cd_{ii} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & b_{11} & b_{12} & b_{13} & b_{14} \\ 0 & a_{22} & a_{23} & a_{24} & 0 & b_{22} & b_{23} & b_{24} \\ 0 & 0 & a_{33} & a_{34} & 0 & 0 & b_{33} & b_{34} \\ 0 & 0 & a_{43} & a_{44} & 0 & 0 & b_{43} & b_{44} \\ c_{11} & c_{12} & c_{13} & c_{14} & d_{11} & d_{12} & d_{13} & d_{14} \\ 0 & c_{22} & c_{23} & c_{24} & 0 & d_{22} & d_{23} & d_{24} \\ 0 & 0 & c_{33} & c_{34} & 0 & 0 & d_{33} & d_{34} \\ 0 & 0 & 0 & c_{44} & 0 & 0 & d_{43} & d_{44} \end{pmatrix}$$

$$(1)$$

We have followed the proposal of Golub, Nash and Van Loan [4] to reduce the cost of triangularizing this matrix $(O(m^3)$ flops[2]). Applying to the problem a permutation matrix such that it transforms $1, 2, \ldots, mn$ into $1, n + 1, 2n + 1 \ldots, (m-1)n+1, 2, n+2, 2n+2, \ldots, (m-1)n+2, \ldots, n, 2n, 3n, \ldots, (m-1)n, mn$ the result is an equivalent problem in which the coefficient matrix is an upper

---

[2] According to the old definition of flops [5], $a[i] = a[i] + b[i] * c[i]$, to better compare the sequential algorithm with the systolic implementation.

triangular matrix with two non-zero subdiagonals. Using that transformation in the example, the result is

$$
P^T \begin{pmatrix} a_{11} & a_{12} & a_{13} & \mathbf{a_{14}} & b_{11} & b_{12} & b_{13} & \mathbf{b_{14}} \\ 0 & a_{22} & a_{23} & \mathbf{a_{24}} & 0 & b_{22} & b_{23} & \mathbf{b_{24}} \\ 0 & 0 & a_{33} & \mathbf{a_{34}} & 0 & 0 & b_{33} & \mathbf{b_{34}} \\ 0 & 0 & a_{43} & \mathbf{a_{44}} & 0 & 0 & b_{43} & \mathbf{b_{44}} \\ c_{11} & c_{12} & c_{13} & \mathbf{c_{14}} & d_{11} & d_{12} & d_{13} & \mathbf{d_{14}} \\ 0 & c_{22} & c_{23} & \mathbf{c_{24}} & 0 & d_{22} & d_{23} & \mathbf{d_{24}} \\ 0 & 0 & c_{33} & \mathbf{c_{34}} & 0 & 0 & d_{33} & \mathbf{d_{34}} \\ 0 & 0 & 0 & \mathbf{c_{44}} & 0 & 0 & d_{43} & \mathbf{d_{44}} \end{pmatrix} P = \begin{pmatrix} a_{11} & b_{11} & a_{12} & b_{12} & a_{13} & b_{13} & \mathbf{a_{14}} & \mathbf{b_{14}} \\ c_{11} & d_{11} & c_{12} & d_{12} & c_{13} & d_{13} & \mathbf{c_{14}} & \mathbf{d_{14}} \\ 0 & 0 & a_{22} & b_{22} & a_{23} & b_{23} & \mathbf{a_{24}} & \mathbf{b_{24}} \\ 0 & 0 & c_{22} & d_{22} & c_{23} & d_{23} & \mathbf{c_{24}} & \mathbf{d_{24}} \\ 0 & 0 & 0 & 0 & a_{33} & b_{33} & \mathbf{a_{34}} & \mathbf{b_{34}} \\ 0 & 0 & 0 & 0 & c_{33} & d_{33} & \mathbf{c_{34}} & \mathbf{d_{34}} \\ 0 & 0 & 0 & 0 & a_{43} & b_{43} & \mathbf{a_{44}} & \mathbf{b_{44}} \\ 0 & 0 & 0 & 0 & 0 & d_{43} & \mathbf{c_{44}} & \mathbf{d_{44}} \end{pmatrix} \quad (2)
$$

Different possibilities were considered when designing the corresponding algorithm to avoid the construction of the auxiliary matrix $P^T \mathcal{M} P$. Two were deeply studied due to their feasibility:

1. To process $\mathcal{M}$ as matrix $(Ab_{ii} + Cd_{ii})$ in the SGH step. The basic idea in the procedure described in figure 2 is to look for a compatible data flow among the operations to allow a systolic implementation. Then the transformation to triangularize the coefficient matrix of **Solve** is applied by columns. In the systolic implementation the resulting data flow allows to obtain a good chaining between **Calculate Q** and **Solve** operations, that stands also for **Solve** and **Gaxpy**; and, moreover, there is no need to form an auxiliary matrix, working in terms of the original one. Our aim was also to keep the original matrices in the **Solve_2** operation, following for the triangularization the reduction order imposed by the permutation of $\mathcal{M}$ in eq. 2. The result was the design of a sequential algorithm, SGG1 [10], of $O(5m^2n + mn^2)$ flops.

2. To process $\mathcal{M}$ in a similar way to the Back Substitution Algorithm, obtaining the values of columns x_i and x_{i-1} by groups of two elements (corresponding to zero subdiagonal elements of matrix A) or four elements (corresponding to non-zero subdiagonals entries of matrix A). That must be done due to the structure of $\mathcal{M}$ in eq. 1. For non-zero entries of the original matrix A (for example elements a_{43}, b_{43} and d_{43}) a 4×4 system has to be solved, obtaining four values of ith and $i - 1^{th}$ columns of X. For entries whose value is zero, solving a 2×2 system two values of ith and $i - 1^{th}$ columns of X. The corresponding sequential algorithm [10] has a temporal cost of $O(m^2n + mn^2)$ flops.

$$
\begin{pmatrix} a_{33} & b_{33} & c_{34} & d_{34} \\ c_{33} & d_{33} & c_{34} & d_{34} \\ a_{43} & b_{43} & a_{44} & b_{44} \\ 0 & d_{43} & c_{44} & d_{44} \end{pmatrix} \begin{pmatrix} x_{3,i-1} \\ x_{3,i} \\ x_{4,i-1} \\ x_{4,i} \end{pmatrix} = \begin{pmatrix} e_{3,i-1} \\ e_{3,i} \\ e_{4,i-1} \\ e_{4,i} \end{pmatrix} \cdot \begin{pmatrix} a_{22} & b_{22} \\ c_{22} & d_{22} \end{pmatrix} \begin{pmatrix} x_{2,i-1} \\ x_{2,i} \end{pmatrix} = \begin{pmatrix} e_{2,i-1} \\ e_{1,i} \end{pmatrix}
$$

3.1 Obtaining Systolic Algorithms for the Solve_2 Operation.

The previous resolution schemes present two major drawbacks for their systolic implementation:

1. For the first approach, the rotations involve columns in different blocks of the original matrices (marked in bold in eq. 2); therefore it is necessary to

explicitly form all the linear combinations of all the blocks involved in the Update of other columns of matrix E. It is impossible to form the auxiliary vectors w1, w2, v1 and v2 to reduce the cost of Update.

2. For the second approach data dependencies are so strong that we could not find an efficient systolic algorithm for it.

Therefore, to design an efficient systolic algorithm for the Solve_2 operation, we studied the reuse of those obtained for simpler cases. When solving the Triangular and the Hessenberg case, two basic systolic arrays were designed [11]. The first one, called **Module QR**, has the capability of performing the operation

$$Calculate \ Q : (\alpha A + \beta B)Q \ is \ upper \ triangular$$

obtaining AQ, BQ and Q, and working with matrices of any size. The description is presented in figure 4. If $\alpha = 1$, $A = A$, $\beta = 0$ and $B = I$, the outputs of this operation are AQ and Q.
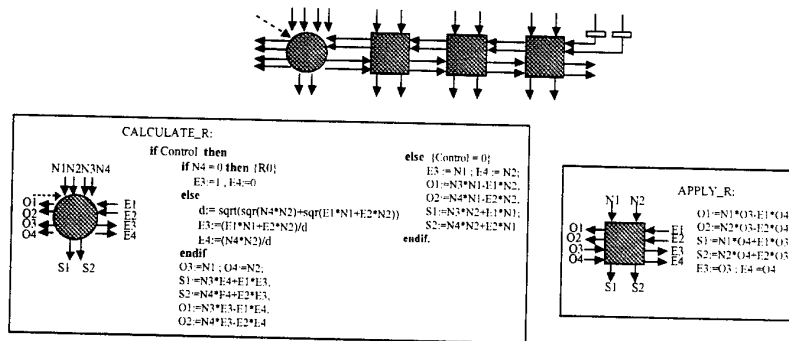

Fig. 4. Module QR.

The second one, called **Module Solve/GAXPY**, has the capability of simultaneously performing the operations

$$Solve \ (\alpha A + \beta B)x = e \ and \ w := A * x, \ v := B * x$$

also working with matrices of any size. The description is presented in figure 5. If $\alpha = 1$, $A = AQ$, $\beta = 0$ and $B = Q$, among the outputs of this operation we have $v = x$, obtained from $x := Q(Q^T x)$.

It is then possible to solve the General case of the Generalized Sylvester Equation using the SGH step when a subdiagonal entry of the matrix D is zero and using the following procedure when a subdiagonal element is non-zero:

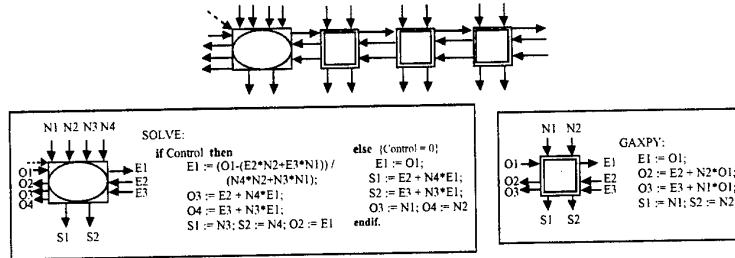1. Construct the $2m \times 2m$ matrix $P^T \mathcal{M} P$,

Fig. 5. Module Solve/GAXPY.

2. Construct the corresponding version of Identity matrix: starting from

$$\mathcal{I} = \begin{pmatrix} I_{m \times m} & I_{m \times m} \\ I_{m \times m} & I_{m \times m} \end{pmatrix}$$

apply on it the same permutation (assuming again m=4),

$$P^T \mathcal{I} P = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}, \tag{3}$$

3. Using the Module QR ($\alpha = 1$, $A = P^T \mathcal{M} P$, $\beta = 0$ and $B = P^T \mathcal{I} P$) nullify the two subdiagonals of matrix $P^T \mathcal{M} P$, $(P^T \mathcal{M} P)Q$ and obtain $(P^T \mathcal{I} P)Q$,
4. Using the Module Solve/GAXPY ($\alpha = 1$, $A = (P^T \mathcal{M} P)Q$, $\beta = 0$ and $B = (P^T \mathcal{I} P)Q$) solve the triangular system and obtain x_i and x_{i-1} from the solution of the system,
5. Using the Module Solve/GAXPY ($A = A$, $B = C$ and any value for α and β) calculate w1, w2, v1 and v2 and Update the matrix E.

This procedure can be entirely implemented with the proposed systolic arrays independently of the size of the coefficient matrices of equation $AXB + CXD = E$.

4 Systolic Implementations for the General Case.

The basic stage of the systolic computation will be the obtaining of a column of matrix X, x_i, when $d_{i-1,i} = 0$ (SGH step) or the obtaining of two columns of matrix X, x_i and x_{i-1}, when $d_{i-1,i} \neq 0$ (SGG step).

Figure 6 shows how to combine the two basic modules to solve the SGH step. In addition to the Module QR and the Module Solve/GAXPY it is needed

a special cell, called GAXPY_2 to complete the calculus of w and v, accumulating on them the corresponding products with the subdiagonal elements of AQ and CQ (with the same zero-structure that matrix A); it is also necessary an array formed by SAXPY cells with capability of performing a Saxpy operation to update each column of matrix E. This update is made up with the value of vector $Q^T x_i$ The figure does not show the calculation of x_i from this value, but it can been performed on the same array, introducing only the Identity matrix, the corresponding rotations and the vector $Q^T x_i$.
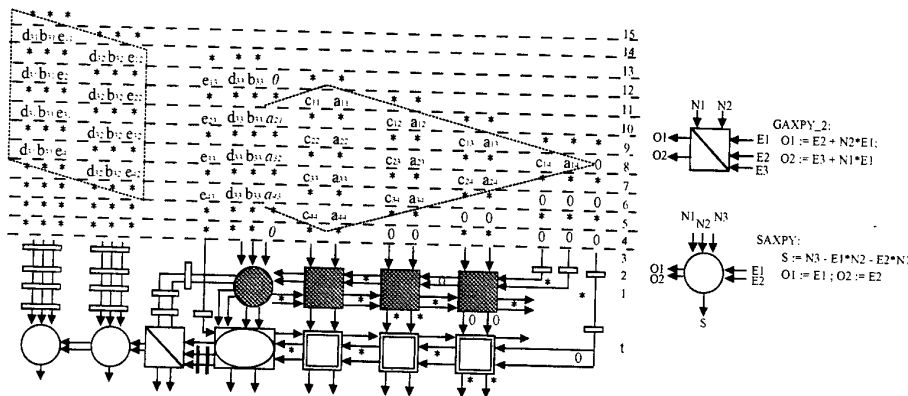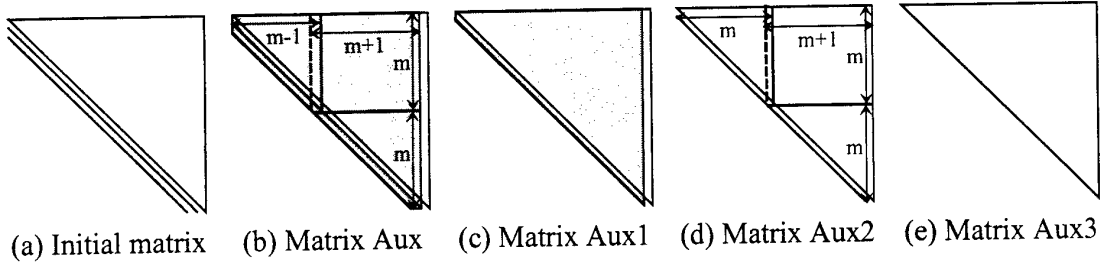

Fig. 6. Obtaining x_i $(d_{i-1,i} = 0)$.

The computation of the SGG step is formed by the successive transformation (fig. 7) of the $2m \times 2m$ matrix $P^T \mathcal{M} P$. To nullify the second subdiagonal, it is considered the matrix Aux, formed only by the $(2m - 1)$ first columns of the original matrix; that is, it is a Hessenberg matrix of size $2m \times (2m - 1)$. When the subdiagonal has been nullified, the matrix $Aux1$, also of size $2m \times (2m - 1)$, is obtained. To form the matrix $Aux2$, it is necessary to add the last column of the initial matrix. Again, a Hessenberg matrix, of size $2m \times 2m$, is obtained and after the process, it is obtained $Aux3$, that is upper triangular.

Figure 8 shows the complete process and the order in which each one of these auxiliary matrices is processed. Note that the modules are of size m, so the process supposes the application of the DBT [14] on these matrices. The DBT of the operation Calculate Q will be more widely discussed in subsection 4.1, but note that, in figure 7, the matrices are cut in blocks of the size of the arrays in a special way, making two blocks share a column.

4.1   Size-Independent Systolic Implementation.

Let us suppose that the blocks system of figure 1 is made up by $N \times N$ upper Schur blocks $Ab_{ij} + Cd_{ij}$, of size $M \times M$, and each block is built of $q \times q$ blocks

(a) Initial matrix   (b) Matrix Aux   (c) Matrix Aux1   (d) Matrix Aux2   (e) Matrix Aux3

Fig. 7. Successive transformation of the matrix $P^T \mathcal{M} P$.

of dimension $m \times m$, being $N = pn$ and $M = qm$. Let us also suppose that each of the columns of X and E will be built of q blocks of size m. According to this block structure, we will identify the subblock at the r row and s column from the $(Ab_{ij} + Cd_{ij})$ block with the notation $(A^{rs}b_{ij} + C^{rs}b_{ij})$; and the rth subvector from the ith column of X, x_i, or E, e_i, will be written x_i^r or e_i^r. This block division will be used to develop a block oriented process to solve the Generalized Sylvester Equation; the described situation allows the decomposition of operations Solve, Gaxpy and Update to process blocks of size $m \times m$. To decompose the operation Calculate Q (and Apply Q) it is necessary to realize that there can exist subdiagonal elements in the matrix $(Ab_{ii} + Cd_{ii})$ that do not belong to any block. In order to nullify them, the block division for this operation is similar to the one depicted in figure 9: two consecutive blocks in a row, $(A^{rs}b_{ii} + C^{rs}d_{ii})$ and $(A^{r,s+1}b_{ii} + C^{r,s+1}d_{ii})$, share a column, in such a way that we can calculate and apply the corresponding rotations.

Also, to perform the Update operation, the following block division for the matrix E and the ith row of matrices B and D must be considered:

$$E = \begin{pmatrix} E_{11} & E_{12} & \cdots & E_{1p} \\ E_{21} & E_{22} & \cdots & E_{2p} \\ \cdots & \cdots & \cdots & \cdots \\ E_{q1} & E_{q2} & \cdots & E_{qp} \end{pmatrix}, \quad \begin{aligned} b_{i,1:1} &= \begin{pmatrix} b_{i1} & b_{i2} & \cdots & b_{iK} & b_{i,K+1} \end{pmatrix} \\ d_{i,1:1} &= \begin{pmatrix} d_{i1} & d_{i2} & \cdots & d_{iK} & d_{i,K+1} \end{pmatrix} \end{aligned}$$

Let us assume K=((i-1) DIV n) and L=((i-1) MOD n). Each block E_{ij} is of size $(m + 1) \times n$, and shares a row with the corresponding block $E_{i+1,j}$. Each subblock of the ith row of B and D has n elements, except for the subblocks $b_{i,K+1}$ and $d_{i,K+1}$ which have L+1.

To solve the problem in the size-independent case, the Dense-to-banded Transformation, DBT [14], has to be applied to the non-triangular submatrices involved in the process. The DBT obtains, from a matrix of size $m \times m$, another one of size $m \times 2m$ or $2m \times m$, but with bandwidth m, by the adequate juxtaposition of the upper and lower triangles of the matrix. In the present problem it is necessary to find a common DBT to all the operations, so the second possibility must be chosen.

As in the size-dependent algorithm, the basic stage will differ depending whether it is found that $d_{i-1,i}$ is zero or not. When $d_{i-1,i} = 0$, the basic stage is
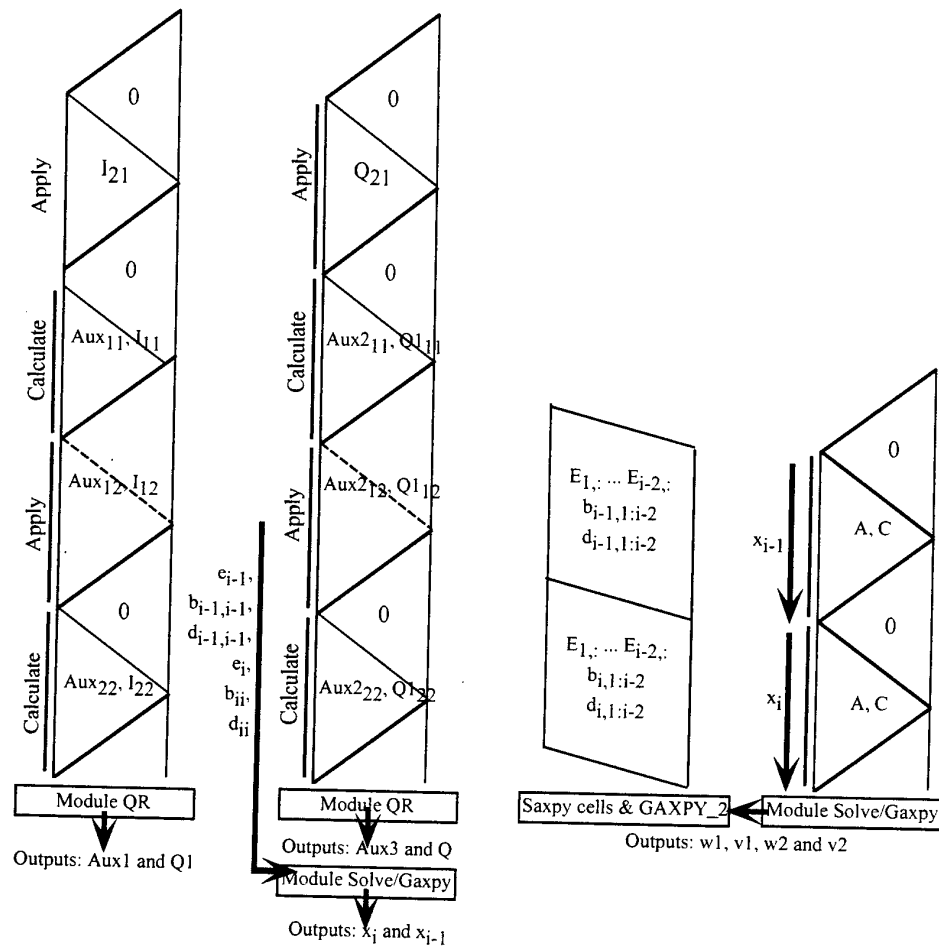
Fig. 8. Successive steps in the calculation of x_i and x_{i-1} and the Update of matrix E. (Note: References to blocks of the Identity and Q matrices really refer to blocks of matrices $P^T \mathcal{I} P$ and $(P^T \mathcal{I} P)Q$ respectively).
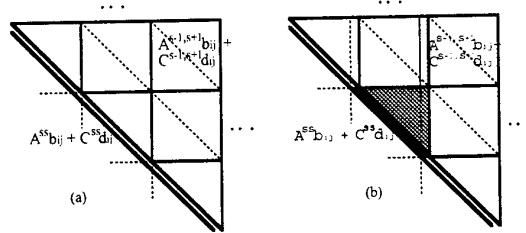
Fig. 9. (a) Block division for the Solve and Gaxpy operations. (b) Block division for Calculate Q and Apply Q operations.

the calculation of x_i^s, shown in figure 10. This process is divided into two steps. First the obtaining of $(Q^s)^T x_i^s$. Then, two different operations on different data are required: the Apply Q and Gaxpy operations to preprocess the w and v vectors for later stages, and the update of E with regard to the calculated value. It is supposed that when obtaining $(Q^s)^T x_i^s$ the control signal is kept high in the QR and Solve/GAXPY modules; afterwards it goes low to start the preprocess, which is developed simultaneously with the updating on the n SAXPY cells array. In the Update operation they will be involved the first L subcolumns of the $E_{s,K+1}$ block and the K first blocks (from E_{sK} to E_{s1}). During this operation, the O(n) array has to receive as inputs the required K copies of w^s and v^s to complete the calculation. To do that, we can use the GAXPY_2 cell: depending of the value of a control signal (independent from the signal managing CALCULATE_Q and SOLVE cell) it selects inputs to the GAXPY array from the SOLVE cell or from memory.

When $d_{i-1,i} \neq 0$ and the Solve_2 operation must be block oriented, the matrix $\mathcal{M}$ must be also divided into blocks; the notation to be used will be:

$$\mathcal{M}^{rs} = \begin{pmatrix} A^{rs}b_{i-1,i-1} + C^{rs}d_{i-1,i-1} & A^{rs}b_{i,i-1} + C^{rs}d_{i,i-1} \\ C^{rs}d_{i-1,i} & A^{rs}b_{ii} +^{rs} Cd_{ii} \end{pmatrix} \tag{4}$$

In this case the basic stage will obtain x_{i-1}^s and x_i^s. Blocks are introduced in the order suggested by figure 10, but taking into account that each diagonal block is processed as shown in figure 8 and each dense block as shown in figure 11. Once x_i y x_{i-1} have been obtained, blocks of matrices A and C are introduced into the array in the order shown by figure 10 to complete the update of blocks $E_{sK}, \ldots, E_{s1}$ while obtaining $w1^s$, $w2^s$, $v1^s$ and $v2^s$.

This theoretical scheme could be optimized in the systolic implementation by overlapping stages, taking profit of the $2-slow$ data flow as well of the existence of operations without data dependencies (for instance, in Solve_2 during the Update of matrix E or, if two consecutive Solve_2 have to be applied, the Update part of the first can be delayed until the beginning of the second, increasing the efficiency of the SAXPY cells).
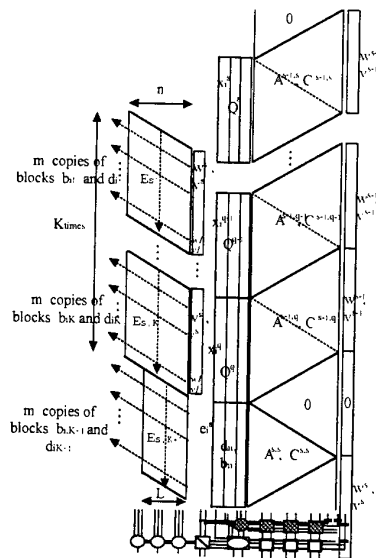
Fig. 10. Data flow for solving x_i^s, m=4,n=3.

5 Conclusions and Future Work

We have shown how the Generalized Sylvester Equation and its derived equations can be systematically solved, using systolic blocks that perform basic operations of the linear algebra, and that form a complete Systolic Library. This method of solving these equations has been obtained by means of a new design methodology. Its main advantage is the modularity of the obtained solution, that allows to apply the same design principles used in software development. The methodology has been applied to other equations derived from that, in the shown cases and in the case of A being a Hessenberg matrix [10], and all of them can be solved with the basic arrays described in this paper. These results have been used to design a complete Systolic Library [11] with the capability of solving a wide variety of problems in the field of matrix algebra.

The work is being further extended in three different directions: the identification of others fields to apply the same design methodology, the implementation of the Systolic Library in FPGA devices and the automation of the process to directly obtain the FPGA configuration from the high level specification of the problem.

References

1. Anderson, E. *et al.*: LAPACK User's Guide. SIAM Publications. Philadelphia. (1992)
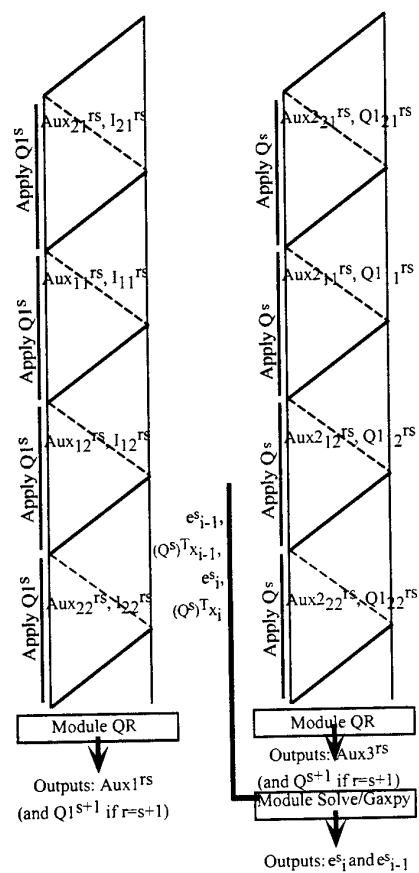
Fig. 11. Detail of the process of each block of submatrix $\mathcal{M}^{rs}, r \neq s$, when applying the previous rotations.

2. Bertseklas, P.H., Tsisiklis, M.M.: Parallel and Distributed Computations. Prentice-Hall Int. Eds. (1989)

3. Cavin, R.K., Bhattacharyya, S.P.: Robust and Well-Condicioned Eigenstructure Assignment via Sylvester's Equation. Optimal Control Applications & Methods, Vol.4 (1983) 205–

4. Golub, G.H., Nash, S., Van Loan, C.F.: A Hessenberg-Schur Method for the Problem $AX + XB = C$. IEEE Trans. on Automatic Control, Vol. AC-24, **6** (1979) 909–

5. Golub, G.H., Van Loan, C.F.: Matrix Computations, 2nd edition. John Hopkins University Press, Baltimore (1989)

6. Lancaster, P.: The Theory of Matrices. Academic-Press, New York (1969)

7. Laub, A.J.: Numerical Linear Algebra Aspects of Control design Computations. IEEE Trans. on Automatic Control, Vol. AC-30, **2** (1985) 97–

8. Lavenier, D., Quinton, P., Rajopadhye, S.: *Advanced Systolic Design*. Digital Systems Processing for Multimedia Systems, chapter 5. Parhi & Nishitani eds. (to appear)

9. Luenberger, D.G.: Time-Invariant descriptor Systems. Automatica, **14** (1978) 473–

10. Martínez, G.: *Algoritmos Sistólicos para la Resolución de Ecuaciones Matriciales Lineales en Sistemas de Control*. Ph.D. Thesis, DSIC, Univ. Politécnica de Valencia, Valencia, Spain (1999)

11. Martínez, G., Fabregat, G., Hernández, V.: *Une Librairie de Routines Systoliques*. Proceedings of Renpar '11 (1999) 181–186.

12. Martínez, G., Fabregat, G., Hernández, V.: A Systolic Library for Solving Matrix Equations. Proceedings of the EUROMICRO'99 Conference (1999) 120–125.

13. Mehrmann, V.: The Autonomous Linear Quadratic Control Problem, Theory and Numerical Solutions. Springer-Verlag, Heidelberg (1991)

14. Navarro, J.J., Llabería, J.M., Valero, M.: Partitioning: an Essential Step in Mapping Algorithms into Systolic Array Processors. IEEE Computer, July (1987) 77–

15. Petkov, P.H., Christov, N.D., Konstantinov, M.M.: Computational Methods for Linear Control Systems. Prentice-Hall, Hertfordshire (1991)

16. Quinton, P., Robert,Y.: *Algorithmes et Architectures Systoliques*. Masson (1989)

17. Sima, V.: Algorithms for Linear-Quadratic Optimization. Marcel Dekker Inc., New York (1996)

18. Van de Geijn, R.: Using PLAPACK: Parallel Linear Algebra Package. MITPRESS (1997), *http://www.cs.utexas.edu/users/plapack*

Parallelizing 2D packing problems with a reconfigurable computing subsystem [*]

J. Carlos Alves[21], C. Albuquerque[2], and J. Canas Ferreira[21] and J. Silva Matos[21]

[1] INESC, Pr. da República, 93 – Apartado 4433 – 4007 PORTO CODEX – Portugal
{jca,cdlalbuq,jcf,jsm}@inescn.pt
[2] Faculdade de Engenharia da Universidade do Porto

Abstract. The 2D packing problem is a NP-hard problem with applications in various industries, from apparel to ship building. Current computer based approaches still rely on interactive software and pick-and-drag procedures performed by experienced people. Semi-automatic commercial systems already exist, but to obtain a final good solution it is still necessary refinements of the solutions obtained automatically. Searching autonomously for good solutions in reasonable execution times requires optimization approaches that rely on the generation and evaluation of a large number of solutions. To accelerate this process, a reconfigurable and parallel computing subsystem was built that works as an auxiliary processor for low-cost desktop PC computers. This paper presents briefly the architecture of the auxiliary processor and the experimental results obtained by different approaches to parallelize the target problem into this parallel architecture.

1 Introduction

The 2D packing problem is a NP-hard problem, consisting in finding a distribution of a given set of irregular shapes over a limited space. Good solutions, although normally sub-optimal, are the ones that lead to minimum waste of the area available for placing the shapes. The particular instance of this problem addressed in this work applies to the textile industry, where the placement area is a width limited rectangular sheet of fabric, and the global objective is to minimize the length of the region used by a particular solution.

Fully automatic approaches targeted to industrial environments must achieve at least the same results as the traditional solutions built by hand, using interactive software applications based on pick-and-drag procedures. Because the NP-hard nature of the problem, it is impossible to guarantee the optimality of one solution. However, good solutions may be found by using meta heuristic search procedures, like local search, tabu search or simulated annealing. These techniques rely on the construction and evaluation of a large number of complete

solutions, in order to guide the search algorithm. General approaches for these techniques start from an initial but feasible solution, and search for better solutions in its vicinity, according to the different criteria used by these procedures. Usually, neighbor solutions are generated by doing elementary modifications in parameters that characterize that solution.

In the 2D packing (or nesting) problem, one solution can be represented by the set of coordinates occupied by the polygons that form the problem data. One neighbor solution may be generated by simply moving one piece a small distance in a certain direction, and re-arranging the others to keep the solution feasible (i.e. avoid overlaps among the polygons). Although this is relatively easy to do by hand with hard paper molds or interactive computer applications, the amount of computation required to perform this operation automatically is too high. In this problem, the critical time consuming tasks are the low level geometric operations that analyze the relative positions between polygons and detect possible overlaps that may turn a solution unfeasible.

To accelerate existing optimization approaches for this problem based on meta-heuristics [1], a custom auxiliary processor for PC computers has been built, based on an array of dedicated processing nodes (PPK—*Polygon Positioning Kernel*) and a programmable processor (FCP—FAFNER *Control Processor*. The PPK nodes are custom digital circuits that perform efficiently the detection of intersections among polygons, thus providing support to handle efficiently the *polygon* datatype. The FCP processor executes a stored program that implements a *nesting heuristic* to build a complete solution, making use of that parallel infrastructure to verify the feasibility of solutions. This custom computing machine is called FAFNER (*Flexible Architecture For NEsting pRoblems*) [2, 3], and interfaces with the higher-level optimization software running in the PC.

This auxiliary processor is built on a reconfigurable digital system based on FPGA circuits (Field Programmable Gate Array). The flexibility afforded with such implementation platform allowed several design iterations on the hardware domain, to experiment with and evaluate different strategies that enabled the efficient exploitation of the computing power available in this system.

This paper presents the results obtained with two heuristic approaches to build solutions for the 2D packing problem, and the different strategies used to parallelize them in the array of PPK nodes. The remainder of this paper is organized as follows. Section 2 detail the core geometric operations involved in this class of problem, and the common procedures that are normally used for the type of application addressed in this work. Section 3 presents the hardware organization of FAFNER, and describes the overall operation of the system. In section 4, different approaches to the parallelization of this problem on the target custom computer and the corresponding results are presented and discussed. Finally, in section 5 the final conclusions are drawn, as well as suggestions for future works.

2 Geometric operations

Although there are various techniques to build solutions for the nesting problem [1, 4, 6, 7, 5], the approach used in this work places polygons on a discrete grid with acceptable size (for example, 1×1 mm is far enough for textile industries), and moves one polygon (the *working polygon*) one grid unit at a time, checking the solution for feasibility for each new position occupied by it. The way polygons are moved and placed into a final and non-overlapping position is defined by a *nesting heuristic*.
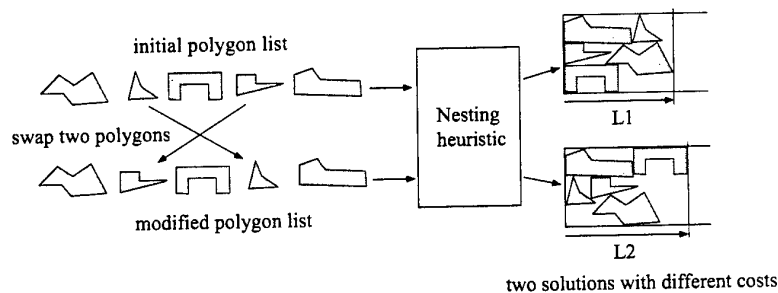

Fig. 1. Different solutions by changing the order of the polygon list.

In this work, one solution is completely specified by an ordered list of polygons (the *polygon list*) and the nesting heuristic that is followed to arrange them (figure 1). Polygons are picked in-order from that list and moved in the placement area, following rules determined by the heuristic. Using the same nesting heuristic, different solutions may be created by doing local modifications in the order of the polygon list. A simple procedure to create neighbor solutions consists in selecting randomly two different polygons in the polygon list and swap their positions. More sophisticated neighbor generation procedures can exploit relationships among different polygons such as area or shape, to favor certain types of solutions.

In what concerns the FAFNER system configured with a given heuristic, one solution is only represented by the polygon list. The FAFNER processor receives this list from the optimization software running in the PC, computes one complete solution and returns the cost of that solution. In the present implementation, the polygon list is represented by a 128 byte vector and the result is one 16 bit integer that measures the length of the rectangular placement area used by that solution. This small amount of data transferred between the host computer and the auxiliary processor for each solution, represents a negligible processing time overhead that is used for data transfer.

The main task of the FCP processor is to move polygons on the placement area, thus implementing the nesting heuristic. To check for feasibility, FCP calls

in parallel the array of PPK nodes. Each PPK node stores locally a list of polygons already placed into their final positions, and verifies the overlap condition between the working polygon and its own list of polygons.

The core operation performed by each PPK node is to verify if the working polygon overlaps each one of its stored polygons. This verification is performed sequentially for all polygons, or until one overlap is found. This verification is done in four phases. First, the relative positions of their bounding boxes are compared. If they do overlap, a more detailed edge-by-edge comparison must be performed. To further speedup this process, edges are grouped into *second-level bounding boxes—SLBB* that are checked first, before comparing pairs of edges of the two polygons. To verify if two edges intersect, their bounding boxes are compared first, and a more complex and time consuming procedure based on D-functions [8] is started only if it is necessary.

This hierarchical procedure saves large amounts of computation. For an industrial problem optimized with a tabu search procedure, near 6,000 millions pairs of polygons are analyzed, but only 7.3% are required for the edge-by-edge analysis. The total number of polygon edges checked for intersection is near 81,000 millions but only 0.6% of this number require the more complex D-function analysis. This problem has 48 polygons, 10 different shapes and a total of 960 edges and has been adopted as the main benchmark used to evaluate the various implementations created in this work.

As a result of this hierarchical procedure, the time spent in each processing step is very different: a bounding box comparison takes only one clock cycle (either for polygons, second level or edges), but to conclude if two edges intersect or not, it may be necessary up to 11 clock cycles. Because of this, the time required to evaluate the overlap condition of two polygons varies from 1 clock cycle to a worst case that must perform the exhaustive edge-by-edge analysis, and exceeds $N \times M \times 11$ clock cycles, where N and M represent the number of edges of each polygon. The actual processing time required for this operation is thus dependent on various factors: the relative position of the polygons, their shape and the number of edges that are associated into SLBBs.

3 Fafner architecture

The FAFNER accelerator system is composed by two processing units: the FCP and the PPK array (figure 2) [2, 3]. The FCP (FAFNER *Control Processor*) is a custom stored program processor that executes a program implementing the nesting heuristic. This architecture uses an instruction set and a memory organization that have been specifically designed for this application, enabling the necessary support to handle a convenient *polygon* datatype. Custom low-level instructions implement the communication with the array of PPK nodes, through a reduced set of commands accepted by them.

The array of PPK nodes constitute a parallel engine dedicated to the evaluation of intersections between polygons. This array communicates with FCP through a common bus, plus an additional circuit that works as a concentrator of
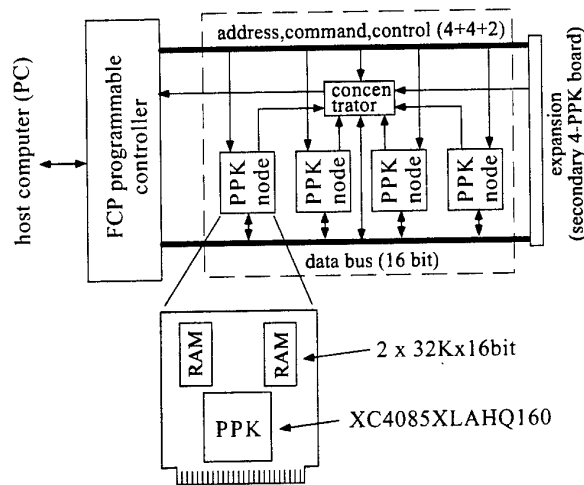
Fig. 2. The FAFNER system architecture.

the responses of each processing node. Each node interprets commands issued by FCP that define processing options, load polygon data into PPK nodes, assign a position for the working polygon and execute the check for overlap procedure against the set of placed polygons stored into the local memory of each PPK node. The addressing mechanism allows commands and data being sent to only one PPK node or broadcast for all PPK nodes present in the array. Present implementation includes 4 processing nodes, although the current version may be extended up to 12 nodes without requiring any modification in the physical hardware system.

The concentrator circuit manages the responses of all the PPK nodes connected to it. When the PPK array is called to check for overlap, each PPK node works in parallel with different sets of data. A direct consequence of this is that, in the general case, each PPK node will terminate its processing within different times, depending on the geometric relationships between the working polygon and the set of polygons it is checked with. Moreover, the results of that processing (either overlap detected or overlap not detected) may be different for each node. The concentrator implements a custom circuit that manages all the status signals output by the PPK nodes, and feed appropriate responses to the FCP processor. The complexity of this function depends on the nesting heuristic and the way it is implemented. This varies from a three 4-input logic gates to a much more complex control circuit that compares the positions of the working polygon in all the PPK nodes to determine which one has found the best feasible position.

The FAFNER system has been implemented in a reconfigurable system built with XILINX FPGA devices [9] and additional memory chips. A library of interface routines has also been created to support the development of application

programs that use this infrastructure. The fast reconfiguration of this family of programmable chips enabled easy and fast design iterations of the hardware system. During this development, this has been crucial to tune up and improve the efficiency of the hardware architecture of both the FCP and PPK, without requiring any modification in the physical hardware platform. Besides, various implementations of the whole system with specific optimizations for different heuristics have been developed and can be programmed in a matter of seconds. Figure 3 shows a picture of the FAFNER system with 4 PPK nodes.
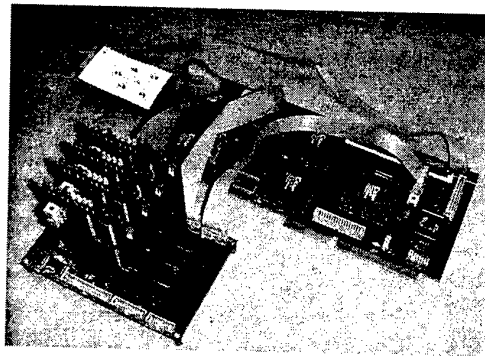

Fig. 3. The FAFNER system with 4 PPK nodes.

4 Parallel approaches to nesting problems

Because the critical time consuming task is the check for overlap operation, one important issue is how to dispatch and schedule these operations by the PPK nodes, in order to exploit efficiently the computing power available in the system. The technique used to build one solution and the strategy adopted to distribute the polygons by the PPK nodes are important factors that largely influence the effective gain in speed by using various PPK processors working in parallel. The various approaches that are being experimented in the scope of this work, and the results obtained are presented in the next subsections. All the techniques implemented represent a solution by an ordered list of polygons, as referred above in section 2.

The benchmark adopted in this work is an industrial problem taken from a textile industry. It is formed by 48 polygons, 10 different shapes and a total of 960 edges. These results were obtained with an implementation of the FAFNER system running at 10 MHz, for a sequence of 20 different solutions generated randomly, creating neighbor solutions by swapping two polygons in the polygon list.

4.1 The right-to-left algorithm

A first nesting heuristic implemented in FCP and described in [1] consists in pushing polygons from right to left, seeking for a leftmost feasible position for each polygon. A new polygon is first positioned into a leftmost position that do not overlap any of the other polygons already placed. Then, it is moved to the left one unit at a time, while checking for overlap with the other placed polygons. If an overlap is found, the previous position is restored and additional up and down moves are tried until a final position is defined for that polygon. Figure 4 illustrates the path followed by one polygon until its final position is reached.
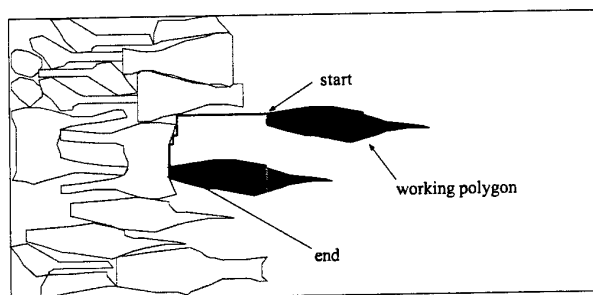

Fig. 4. The right-to-left nesting heuristic.

To parallelize the evaluation of intersections by the PPK nodes, the set of polygons already placed is distributed evenly by all the nodes. When the algorithm run by FCP needs to verify the intersection of a new polygon against all the placed polygons, the check for overlap command is broadcast for all the PPK nodes. This operation terminates as soon as one node detects an intersection, or when all the nodes conclude the processing without finding any overlap with their own set of polygons. When a final position is established for the working polygon, it is stored in the local memory of a PPK node selected by the nesting heuristic. In this approach, this is done in a cyclic fashion to distribute them evenly by all the PPK nodes.

With this strategy, the complex operation that checks one polygon for overlap against a set of polygons is well distributed by all the PPK nodes, each one working in parallel with disjoint sets of polygons. As shown in the example of figure 5, when the 16 check for overlap operations have to be performed, this strategy divides the number of check for overlap operations by the number of PPK nodes. If these operations require approximate processing times, this procedure will also divide the global processing time by the number of processing nodes available in the system.

However, practical results have shown little improvements in the overall performance when using four instead of one PPK node. Further analysis of these
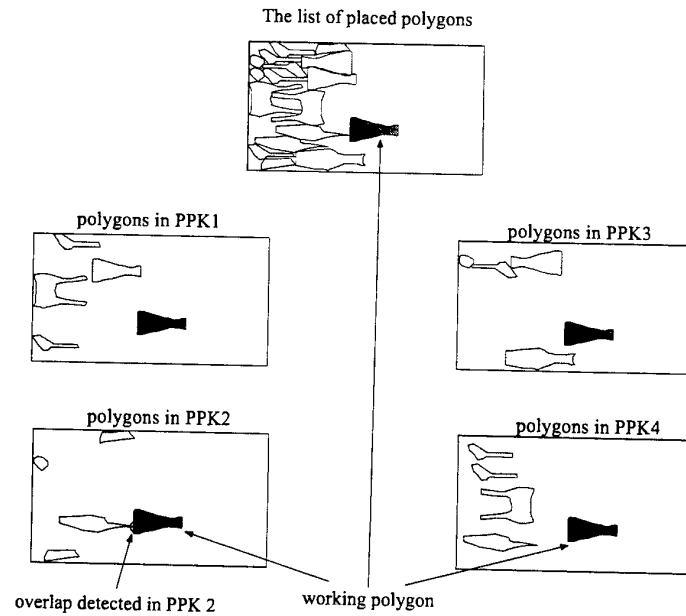
The list of placed polygons

Fig. 5. The parallelization strategy in the right-to-left nesting heuristic.

results have shown that this is due to the disparity of processing times required for the evaluation of intersections, as it was referred above. While the 4 PPK nodes actually start their computation in parallel, in the majority of cases only one node requires the complex edge by edge analysis, and all the other PPKs terminate their processing based only on the analysis of bounding boxes, within a few number of clock cycles. In this situation, the overall processing time is clearly dominated by the work of a single node that performs the complex edge-by-edge analysis, or even by the code run in FCP when all nodes conclude their processing by analyzing only bounding boxes, either of polygons, SLBBs or edges.

Table 1 presents the average execution times required by the combined hardware/software system to build one complete solution. The slight 6.8% reduction in the execution time achieved by using 4 PPK nodes instead of a single node does not justify the investment of the additional processing nodes. This improvement is even reduced to almost zero for simpler benchmarks, where the software run in the PC and in FCP far dominates the global processing time.

This procedure creates solutions of the nesting problem with short execution times, because most cases of the critical check for overlap operations are determined by analysis of bounding box. However, because backtracking and unfeasible solutions are not allowed during the positioning of polygons, there are severe limitations that constrain the quality of the solutions generated. For example, smaller polygons cannot travel over larger polygons to be placed in

Table 1. Execution times for the right-to-left algorithm

Number of PPK nodes	4	3	2	1
Execution time (sec)	0.463	0.472	0.478	0.497
Improvement (%)	6.8%	5.1%	3.8%	—

blank areas that could be left between the larger polygons. Because of these limitations and the bad utilization of the parallel array of PPK nodes, other nesting heuristics were implemented that achieved much better results.

4.2 The raster algorithm

Another approach is based on the algorithm referred in [5]. This nesting heuristic also places one polygon at a time, but searches exhaustively the placement area to find the leftmost position were the moving polygon may be placed. This technique solves the problems referred above and yields much better results in terms of quality, exploiting better the parallelism of the PPK array.

A new polygon is placed first in the upper left corner of the placement area, and a top to bottom, left to right raster is performed one unit at a time, until a feasible position is found. To avoid unnecessary steps and speedup the overall processing, the starting position of a new polygon may be set to the final position found for the last polygon with the same shape. This technique follows the same procedure to distribute the evaluation of intersections by the processing nodes. The FCP processor manages the movement of the working polygon on the placement area, and calls the array of PPK nodes to determine if there is any overlap. However, because the path followed by a polygon requires more frequently the more complex edge analysis against polygons stored into different PPK nodes, the average improvement in the execution time achieved with 4 PPK nodes is increased to 12%, using a cyclic distribution of the polygons by the PPK nodes. The operation of this nesting heuristic is illustrated in figure 6, and the results obtained are presented in table 2.

Table 2. Execution times for the raster algorithm.

Number of PPK nodes	4	3	2	1
Execution time (sec)	53.3	56.0	58.0	60.6
Improvement (%)	12.0%	7.6%	4.2%	—

Although the processing times required by this heuristic to build solutions are more than 115 times worst than the previous approach, the quality achieved by these solutions is much better. In most situations, the first solution found by this technique is already better than the best solution encountered with the previous approach after a large number of iterations, typically never below 1000. With this heuristic embedded in a simulated annealing search procedure, a new
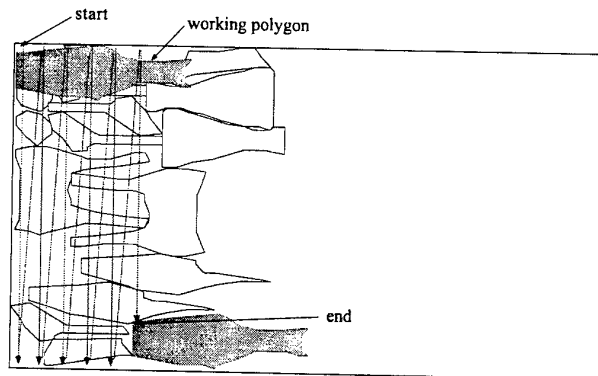
Fig. 6. The raster nesting heuristic.

optimal was found for a synthetic benchmark first proposed in [1] and commonly used to evaluate 2D packing algorithms.

4.3 Refinements to the raster algorithm

In order to speedup the previous approach, the FAFNER architecture was re-designed to move into the PPK nodes functionalities previously accomplished by software run in FCP. The increment of the coordinates that define the position of the working polygon to perform the raster movement was implement in the PPK nodes as dedicated logic circuits based on binary counters and comparators. This added a very small complexity to the PPK nodes, but enabled a significant reduction in the number of instructions executed by FCP in the cycle that searches for the final position of one polygon. With this move into the hardware domain, the main cycle executed by FCP just need to issue a sequence of check for overlap instructions to the PPK array and analyze the results. Each PPK node automatically increments the current position of the working polygon in a single system clock cycle.

As the heuristic procedure is the same, the improvements in the execution times obtained with this refinement are due only to the reduction of the number of instruction in the main loop executed by FCP.

Table 3 presents the results obtained with this implementation. The overall execution times were reduced to approximately 50% of the previous implementation, and the execution time with 4 PPK nodes is reduced by more than 28% when compared to a single PPK node.

4.4 A new approach to the parallelization of the raster algorithm

In spite of the improvement achieved with the previous implementation, the utilization of the PPK processing node is far away from the ideal. In that implementation, the lack of efficiency with the number of processing nodes is also

Table 3. Execution times for the improved raster algorithm

Number of PPK nodes	4	3	2	1
Execution time (sec)	24.0	25.1	27.1	33.7
Improvement (%)	28.8%	25.6%	19.6%	—

related to the disparity of execution times each PPK takes to detect the overlap condition. Contrary to the method used in the first heuristic (see section 4.1), the raster heuristic moves polygons over unfeasible positions, thus requiring in most cases the lower level and time consuming edge-by-edge analysis. Because all PPK nodes are started at the same time to compute the overlap condition with the working polygon in the same position, the final result can only be determined when all nodes conclude their processing. If one or more nodes terminate in a short time because they didn't found any overlap, they must be kept in a idle state to wait for the completion of the slowest node. After that, a new check for overlap may be initiated in the next position.

To further improve the utilization of the PPK array, the complete loop that issues the check for overlap operations to the PPK array was also moved into hardware and implemented in the PPK nodes. This way, each PPK node can search autonomously a final position for the working polygon, terminating only when that position is found or when an external interrupt signal aborts its operation.

To make use of this functionality, the strategy to distribute the list of placed polygons disjointly by the PPK nodes cannot be used. In this implementation, each PPK node holds the complete list of placed polygons, and each node checks the feasibility for disjoint sets of discrete coordinates. The scheme implemented currently for a FAFNER system with N PPK nodes, places initially the working polygon in positions (X, Y) for node 1, $(X, Y + 1)$ for node 2 and $(X, Y + N)$ for node N, and each PPK increments automatically its Y coordinate N units at a time. Within this increment, the Y coordinate is compared with the maximum width defined for the placement area to adjust the X coordinate accordingly.

With this implementation, the FCP processor defines only the initial position for the working polygon, issues the check for overlap operation to the PPK array and polls a status port from the PPK array to wait for the end of computation. When one PPK node finds a feasible position, that position can only be accepted by FCP if all the other nodes are beyond that position. In this case, the PPK nodes still working are interrupted to abort their operations, and the working polygon is frozen in that position and stored into a PPK node determined by FCP. If one PPK node encounters one position but there is at least one of the other nodes behind that position, the array must keep the normal processing until that position is overtaken. This is necessary because a better position may be found by the PPKs that are still working. The management of the responses from the PPK nodes is done by the concentrator circuit. This includes a custom controller that keeps track of the (X, Y) coordinates currently present in each node and decides whether a feasible solution found by one node may be

accepted or not. Whenever a final position is accepted, the concentrator sends an interrupt signal to all PPK nodes and informs the FCP which one has found that position. This is necessary to retrieve from that PPK the actual (X, Y) coordinates occupied by the working polygon.

Table 4. Execution times for the new raster algorithm

Number of PPK nodes	4	3	2	1
Execution time (sec)	12.8	16.9	25.3	50.6
Improvement (%)	74.7%	66.7%	50.1%	—

As the results presented in table 4 show, this implementation of the raster algorithm enabled an optimal balance of the computation load by all the processing nodes. The variation of processing times for different numbers of processing nodes shows a linear increase of performance, measured as number of solutions per unit of time.

5 Conclusions and future work

This paper presented the first results obtained with different approaches for the parallelization of the 2D packing problem into a custom computing machine. The best implementation obtained so far (section 4.4) has achieved, for a real industrial benchmark, a linear performance with the number of processing nodes.

Although the present prototype system only has 4 processing nodes and runs with a slow 10 MHz clock, it performs more than 10 times faster then a present day Pentium processor, running a equivalent software implementation. A new version of this architecture based on last generation FPGAs chips could easily reach a system clock of 20 MHz. Using a FAFNER system populated with 16 PPK nodes, the processing times would be reduced to 1/8. Moving this architecture to a custom integrated circuit technology would further increase the system clock frequency, even though with the cost of loosing the reconfigurability feature afforded by the FPGA-based system.

Future developments will now be focused on the development and evaluation of new strategies to generate neighbor solutions, and to tune up the control parameters used by optimization meta-heuristics. Once the hardware architecture has been settled down, a new version of the FAFNER system will be developed, either based on last generation FPGA chips or custom integrated circuits. This is necessary to make this auxiliary processor an effective tool in accelerating optimization procedures for the 2D packing problem, and to enable the complete automation of the cutting plan phase in textile industries.

References

1. J. F. Oliveira, *Problemas de Posicionamento de Figuras Irregulares: Uma Perspectiva de Optimização*. PhD thesis, Faculdade de Engenharia da Universidade do Porto, 1995.
2. João C. Ferreira, José C. Alves, Célio Albuquerque, José F. Oliveira, José S. Ferreira and José S. Matos, "Flexible hardware acceleration for nesting problems" in *Proceedings of the 5th IEEE International Conference on Electronics, Circuits and Systems*, Vol I, pp. 345-348, Lisboa, Portugal, September 1998.
3. J. Carlos Alves, J. Canas Ferreira, C. Albuquerque, José F. Oliveira, J. Soeiro Ferreira and J. Silva Matos, "FAFNER - Accelerating Nesting Problems with FPGAs" in *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99)* (K. L. Pocek and J. Arnold, eds.), 1999.
4. António M. Gomes, *Novas Contribuições para o Problema de Posicionamento de Figuras Irregulares*. Msc thesis, Faculdade de Engenharia da Universidade do Porto, 1995.
5. S. Segenreich, L. Braga, "Optimal Nesting of General Plane Figures: A Monte Carlo Heuristical Approach" in *Computer&Graphics, Vol. 10 No. 3, pp 229-237, 1986*
6. K. Dowsland, W. Dowsland, "Solution approaches to Irregular Nesting Problems" in *European Journal of Operational Research 84 pp 506-521, 1995*
7. A. Albano, G. Sapuppo, "Optimal Allocation of Two-Dimensional Irregular Shapes Using Heuristic Search Methods" in *IEEE Transactions on Systems. MAN and Cybernetics Vol. SMC-10 No. 5 May 1980*
8. M. Konopasek, "Mathematical treatments of some apparel marking and cutting problems" tech. rep., U.S. Department of Commerce, 1981.
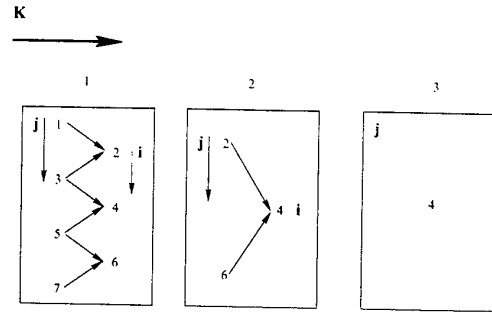9. http//:www.xilinx.com

Fig. 2. the vectors $\mathbf{g}^{(j)}$ are calculated at each iteration of step 3 for $N = 16$ equations.

4 Evaluation

The recursive decoupling algorithm has been implemented on the Fujitsu AP3000 distributed memory computer [13] using the message passing programming model. We have used the MPI programming environment. To verify the performance of the parallel algorithm, we used a test diagonal system (with know solution), whose coefficients matrices satisfy the condition, $|b_i| \geq |a_i| + |c_i|$, $\forall i = 0, 1, ..., N - 1$. This test is described below,

$$
\begin{pmatrix}
2 & -1 & & & & & \\
-1 & 2 & -1 & & & & \\
& -1 & 2 & -1 & & & \\
& & \cdot & \cdot & \cdot & & \\
& & & \cdot & \cdot & \cdot & \\
& & & & -1 & 2 & -1 \\
& & & & & -1 & 2
\end{pmatrix}
\begin{pmatrix}
u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-2} \\ u_{N-1}
\end{pmatrix}
=
\begin{pmatrix}
1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0
\end{pmatrix}
\tag{16}
$$

whose exact solution is an N-dimensional vector $\mathbf{u}$ with components:

$$
u_i = \frac{N + 1 - i}{N + 1}, \ \forall i = 1, \cdots, N.
\tag{17}
$$

The experiments were performed on matrices of size ranging from 16384 (2^{14}) to 1048576 (2^{20}) for the test (16). As we can see in Table 1, the increasing number of processors produces a reduction in the execution time of the algorithm. We observe that this method presents a high efficiency for all the sizes of equations.

Fig. 3 depicts the experimental results. So, in Fig. 3.a we show the efficiency of the modified sequential algorithm we propose related to the initial algorithm efficiency. Thus, Observe than performance increases more than 91% for any value of N. On the other hand, in Fig. 3.b we show the efficiency for the parallel algorihtm for some values of parameter N. Efficiency was calculated using the execution time of the sequential code. The parallel algorithm exceeds the ideal speedup due to an efficient use of local memories and the communication